L Number	Hits	Search Text	DB	Time stamp
1	522955	matrix or matrices	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/09/30 10:51
2	1115	708/446,490,520-525.ccls.	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/09/30 10:51
3	232	(matrix or matrices) and 708/446,490,520-525.ccls.	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/09/30 10:52
4	83	(multipl$7.ti. or multipl$7.ab.) and ((matrix or matrices) and 708/446,490,520-525.ccls.)	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/09/30 10:52
-	22	PMADD	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/06/23 18:00
-	270792	(shape adj adaptive) or SA or SA-DCT	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/06/23 18:06
-	876	((shape adj adaptive) or SA or SA-DCT) and cosine	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/06/23 18:01
-	1036	((shape adj adaptive) or SA or SA-DCT) and (cosine or DCT)	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/06/23 18:01
-	0	(((shape adj adaptive) or SA or SA-DCT) and (cosine or DCT)) and PMADD	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/06/23 18:02
-	55	(((shape adj adaptive) or SA or SA-DCT) and (cosine or DCT)) and SIMD	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/06/23 18:02
-	12849	(((shape adj adaptive) or SA or SA-DCT).ab. or ((shape adj adaptive) or SA or SA-DCT).ti.)	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/06/23 18:07
-	0	((((shape adj adaptive) or SA or SA-DCT).ab. or ((shape adj adaptive) or SA or SA-DCT).ti.)) and (cosine or DCT or transform$7) and SIMD	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/06/23 18:03
-	339	((((shape adj adaptive) or SA or SA-DCT).ab. or ((shape adj adaptive) or SA or SA-DCT).ti.)) and (cosine or DCT or transform$7)	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/06/23 18:03
-	15	PMADDWD	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/06/23 18:04

-	12	PMADDWD and (cosine or DCT or transform$7)	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/06/23 18:05
-	1	SIMD and (PMADDWD and (cosine or DCT or transform$7))	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/06/23 18:05
-	190	((shape adj adaptive) or SA or SA-DCT).ti.	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/06/23 18:06
-	0	(((shape adj adaptive) or SA or SA-DCT).ti.) and 708/$.ccls.	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/06/23 18:06
-	18	((((shape adj adaptive) or SA or SA-DCT).ab. or ((shape adj adaptive) or SA or SA-DCT).ti.)) and 708/$.ccls.	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/06/23 18:11
-	0	((((shape adj adaptive) or SA-DCT).ab. or ((shape adj adaptive) or SA-DCT).ti.)) and 708/$.ccls.	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/06/23 18:11
-	53	(((shape adj adaptive) or SA-DCT).ab. or ((shape adj adaptive) or SA-DCT).ti.)	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/06/23 18:12
-	0	2001339727.URPN.	USPAT	2003/06/23 18:14
-	2	5990956.URPN.	USPAT	2003/06/23 18:18
-	0	SA-IDCT	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/06/24 16:14
-	199	708/402.ccls.	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/06/24 16:14
-	173	708/402.ccls. and (inverse$1 or IDCT)	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/06/25 08:59
-	5	MMX and PMADDWD	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/06/25 09:10
-	1	SIMD and (MMX and PMADDWD)	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/06/25 09:11

US006490607B1

(12) **United States Patent**
Oberman

(10) **Patent No.: US 6,490,607 B1**
(45) **Date of Patent: Dec. 3, 2002**

(54) **SHARED FP AND SIMD 3D MULTIPLIER**

(75) Inventor: **Stuart F. Oberman**, Sunnyvale, CA (US)

(73) Assignee: **Advanced Micro Devices, Inc.**, Sunnyvale, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/416,401**

(22) Filed: **Oct. 12, 1999**

Related U.S. Application Data

(63) Continuation-in-part of application No. 09/014,454, filed on Jan. 1, 1998, now Pat. No. 6,144,980, which is a continuation-in-part of application No. 09/049,752, filed on Mar. 27, 1998, now Pat. No. 6,269,384, which is a continuation-in-part of application No. 09/134,171, filed on Aug. 14, 1998, now Pat. No. 6,223,198.

(51) **Int. Cl.**[7] **G06F 7/52**
(52) **U.S. Cl.** **708/620**; 708/503; 708/498
(58) **Field of Search** 708/620, 503, 708/497, 498

(56) **References Cited**

U.S. PATENT DOCUMENTS

4,620,292	A *	10/1986	Hagiwara et al.	708/503
4,953,119	A *	8/1990	Wong et al.	708/503
5,293,558	A *	3/1994	Narita et al.	708/620
5,675,526	A *	10/1997	Peleg et al.	708/620
6,038,583	A *	3/2000	Oberman	708/620
6,099,158	A *	8/2000	Gorshtein et al.	708/503
6,226,737	B1 *	5/2001	Elliott et al.	708/503
6,233,595	B1 *	5/2001	Cheng et al.	708/503

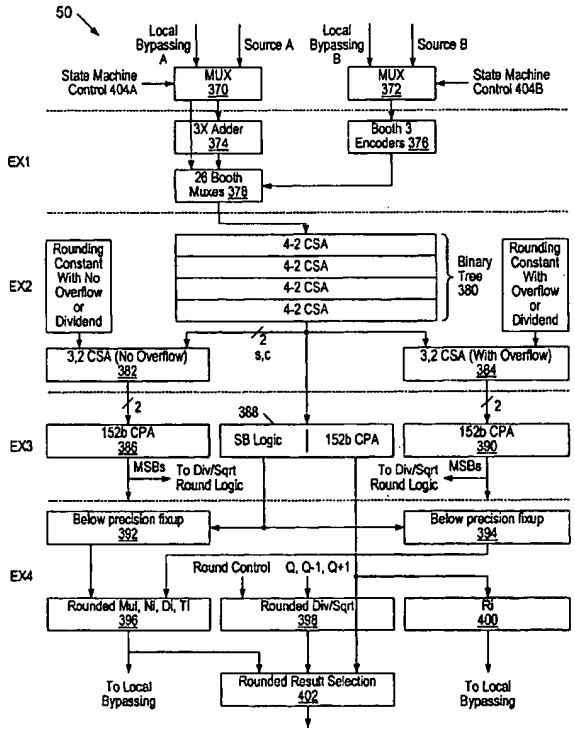
* cited by examiner

Primary Examiner—David H. Malzahn
(74) *Attorney, Agent, or Firm*—B. Noël Kivlin

(57) **ABSTRACT**

A multiplier configured to perform multiplication of both scalar floating point values (X×Y) and packed floating point values (i.e., $X_1 \times Y_1$ and $X_2 \times Y_2$). In addition, the multiplier may be configured to calculate X×Y−Z. The multiplier comprises selection logic for selecting source operands, a partial product generator, an adder tree, and two or more adders configured to sum the results from the adder tree to achieve a final result. The multiplier may also be configured to perform iterative multiplication operations to implement such arithmetical operations such as division and square root. The multiplier may be configured to generate two versions of the final result, one assuming there is an overflow, and another assuming there is not an overflow. A computer system and method for performing multiplication are also disclosed.

20 Claims, 11 Drawing Sheets



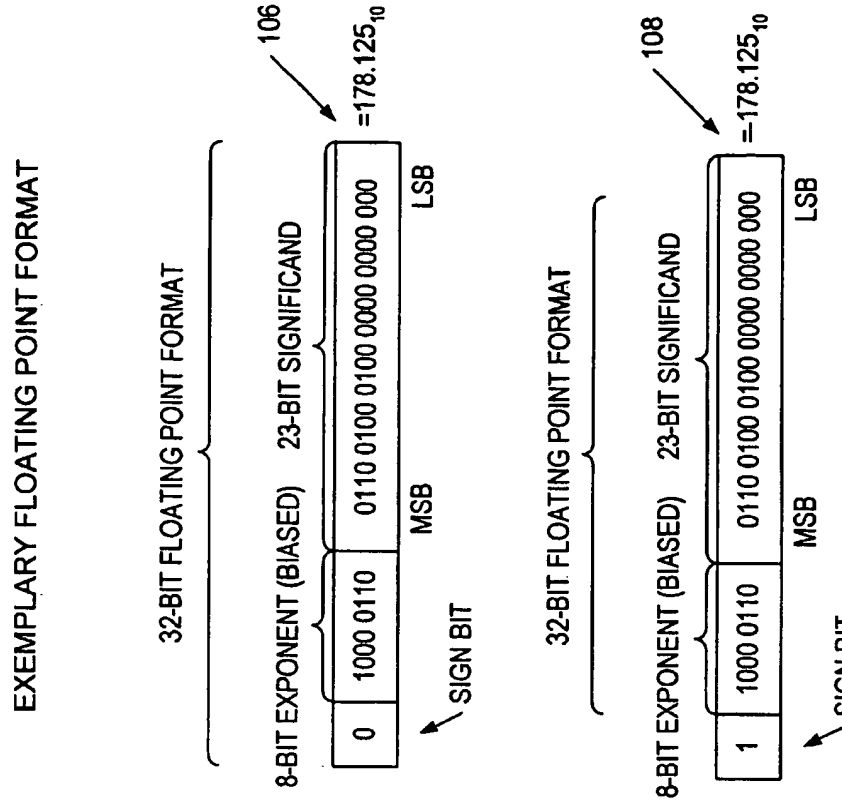

EXEMPLARY FLOATING POINT FORMAT
32-BIT FLOATING POINT FORMAT
106
=178.125$_{10}$
8-BIT EXPONENT (BIASED)
23-BIT SIGNIFICAND
0 1000 0110 0110 0100 0100 0000 0000 000
MSB
LSB
SIGN BIT
32-BIT FLOATING POINT FORMAT
108
=-178.125$_{10}$
8-BIT EXPONENT (BIASED)
23-BIT SIGNIFICAND
1 1000 0110 0110 0100 0100 0000 0000 000
MSB
LSB
SIGN BIT

FIG. 1B

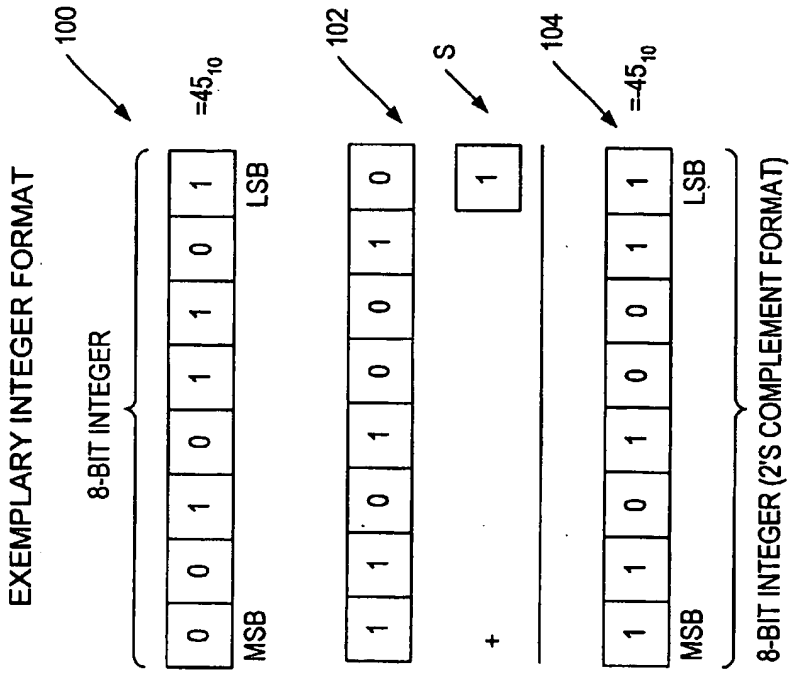

EXEMPLARY INTEGER FORMAT
8-BIT INTEGER
100
=45$_{10}$
0 0 1 0 1 1 0 1
MSB
LSB
102
1 1 0 1 0 0 1 0
S
1
+
104
=-45$_{10}$
1 1 0 1 0 0 1 1
MSB
LSB
8-BIT INTEGER (2'S COMPLEMENT FORMAT)

FIG. 1A

VALUE	SIGN	EXPONENT	SIGNIFICAND
zero	x	$00\ldots00_2$	$0.00\ldots00_2$
infinity	x	$11\ldots11_2$	$1.00\ldots00_2$
QNaN	x	$11\ldots11_2$	$1.1xx\ldots xx_2$
SNaN	x	$11\ldots11_2$	$1.0xx\ldots xx_2$
denormal	x	$00\ldots00_2$	$0.xx\ldots xx_2$

FIG. 2

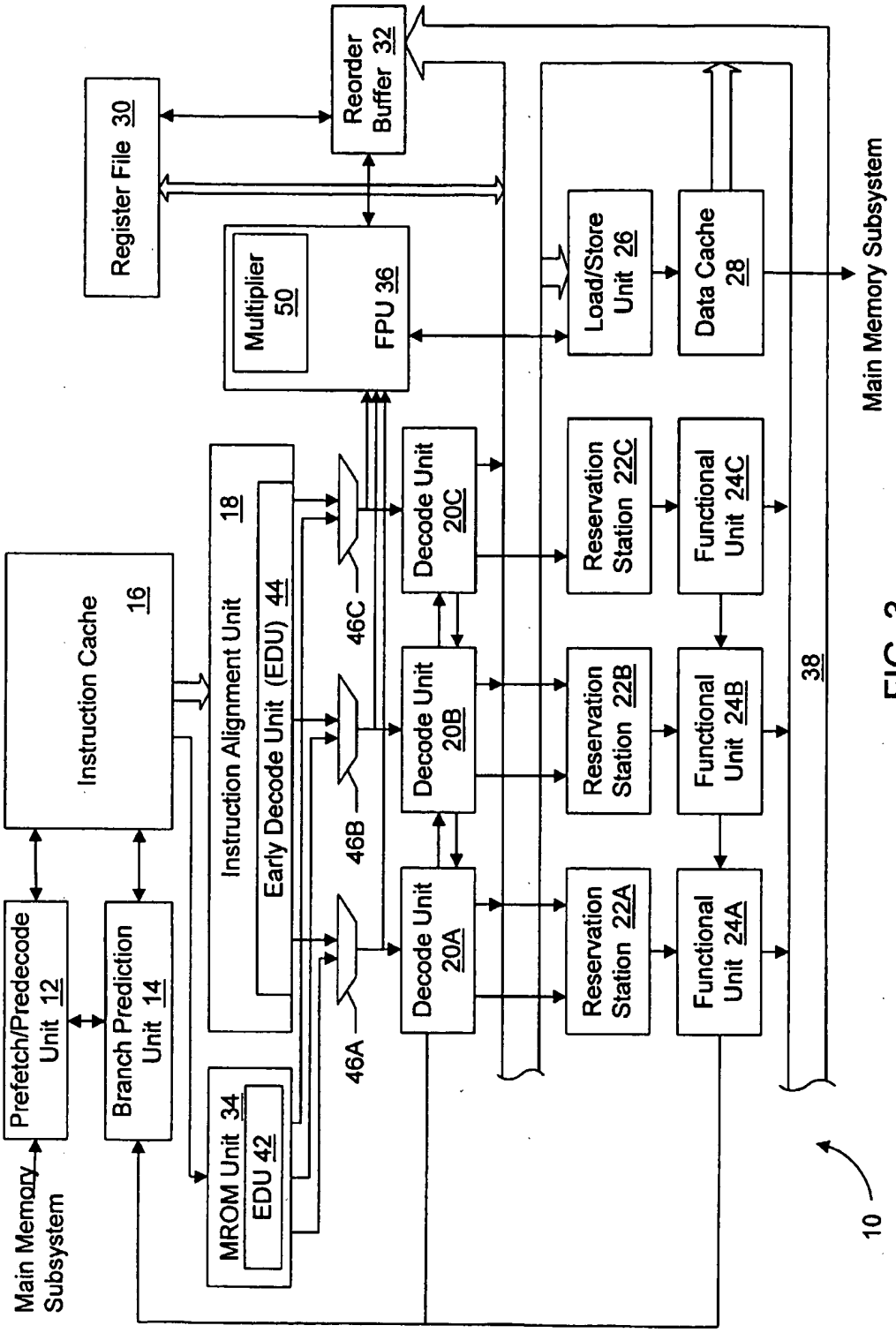

Reorder Buffer 32
Register File 30
Multiplier 50
FPU 36
Load/Store Unit 26
Data Cache 28
Main Memory Subsystem
Instruction Alignment Unit 18
Early Decode Unit (EDU) 44
Instruction Cache 16
Decode Unit 20C
Decode Unit 20B
Decode Unit 20A
Reservation Station 22C
Reservation Station 22B
Reservation Station 22A
Functional Unit 24C
Functional Unit 24B
Functional Unit 24A
46C
46B
46A
38
Prefetch/Predecode Unit 12
Branch Prediction Unit 14
MROM Unit 34
EDU 42
Main Memory Subsystem
10

FIG. 3

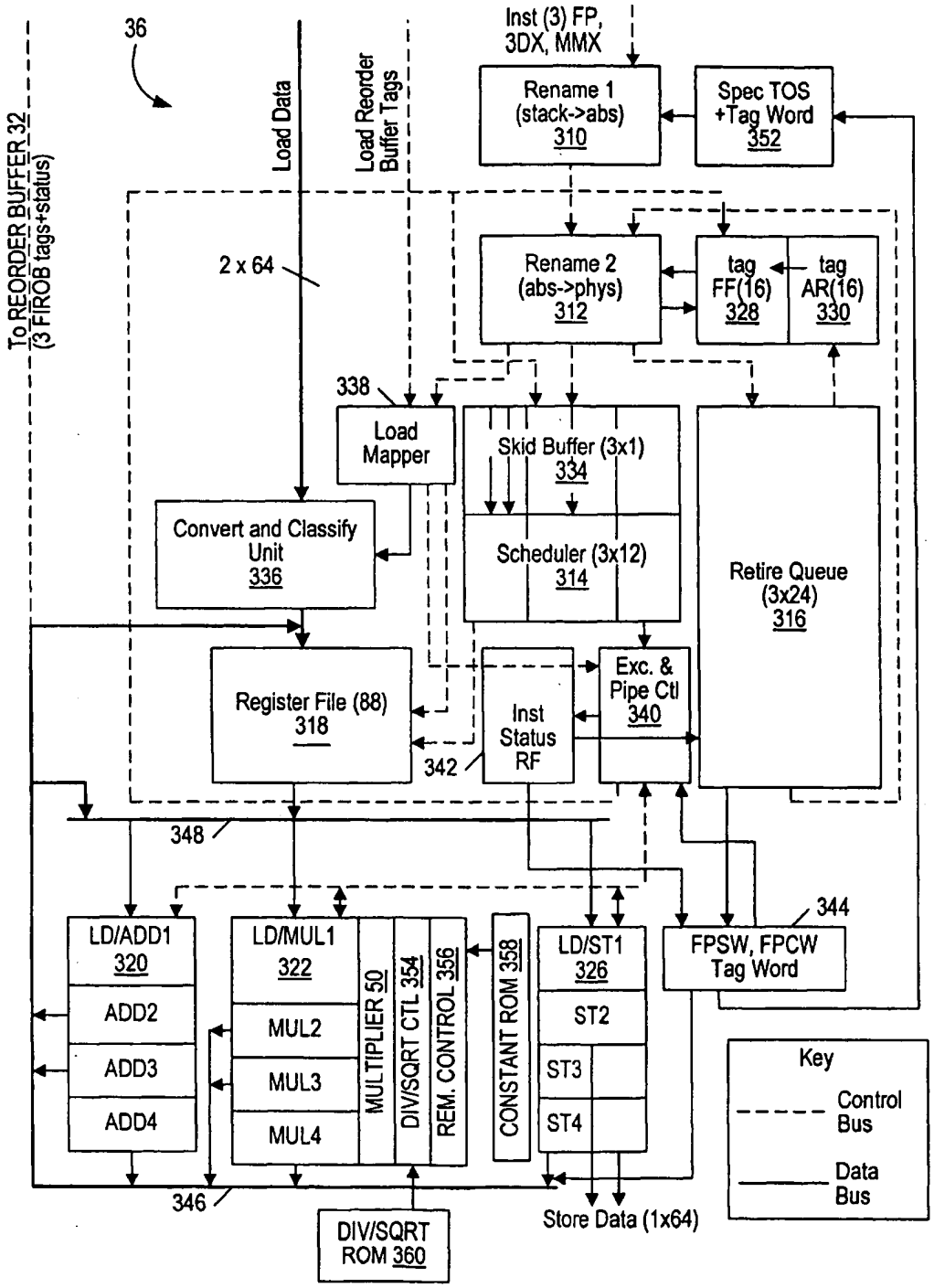

36
Inst (3) FP, 3DX, MMX
Rename 1 (stack->abs) 310
Spec TOS +Tag Word 352
Load Data
Load Reorder Buffer Tags
To REORDER BUFFER 32 (3 FIROB tags+status)
2 x 64
Rename 2 (abs->phys) 312
tag FF(16) 328
tag AR(16) 330
338
Load Mapper
Skid Buffer (3x1) 334
Scheduler (3x12) 314
Retire Queue (3x24) 316
Convert and Classify Unit 336
Register File (88) 318
Inst Status RF
Exc. & Pipe Ctl 340
342
348
344
LD/ADD1 320
ADD2
ADD3
ADD4
LD/MUL1 322
MUL2
MUL3
MUL4
MULTIPLIER 50
DIV/SQRT CTL 354
REM. CONTROL 356
CONSTANT ROM 358
LD/ST1 326
ST2
ST3
ST4
FPSW, FPCW Tag Word
Key
Control Bus
Data Bus
346
DIV/SQRT ROM 360
Store Data (1x64)

FIG. 4

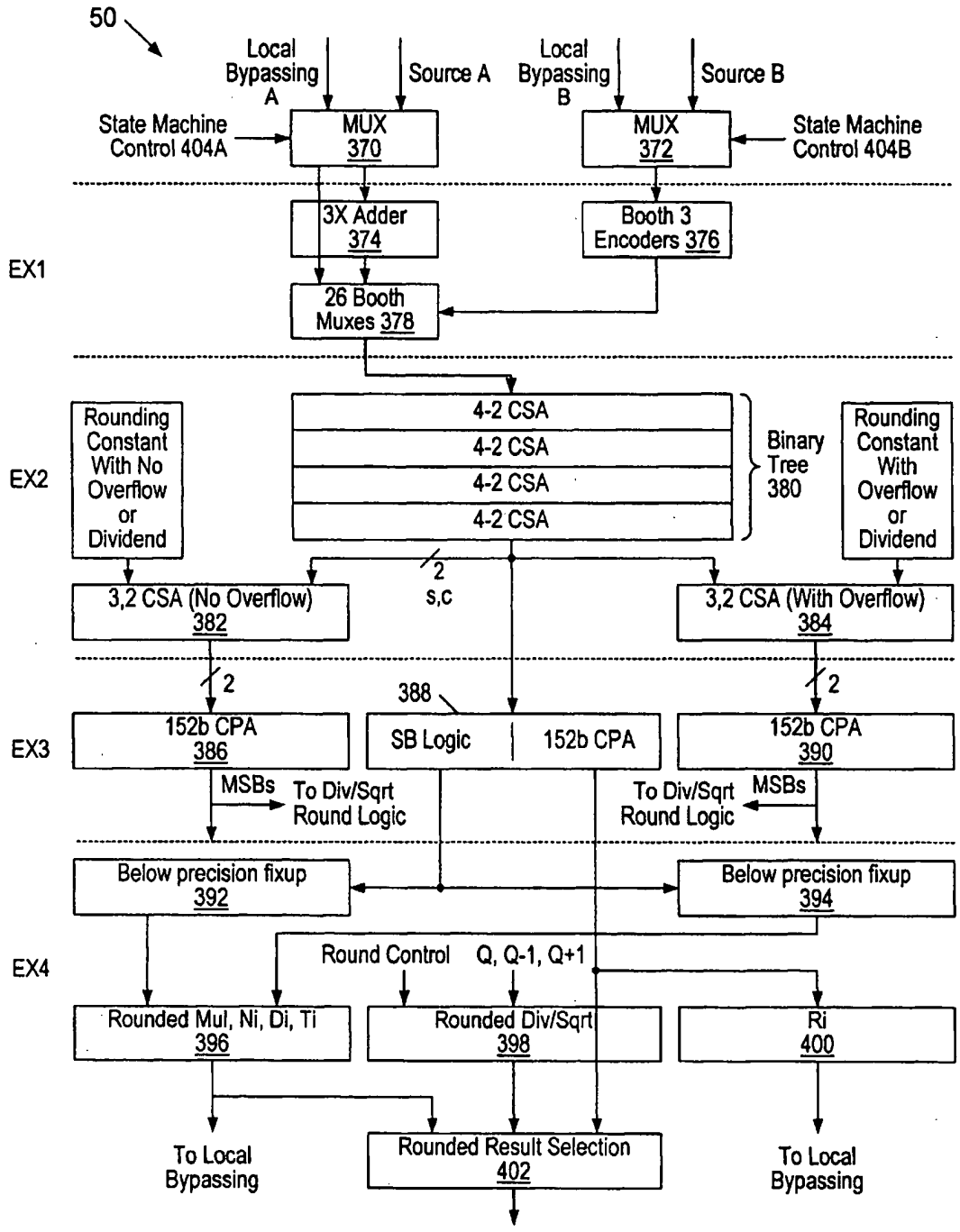

50
Local Bypassing A
Source A
Local Bypassing B
Source B
State Machine Control 404A
MUX 370
MUX 372
State Machine Control 404B
EX1
3X Adder 374
Booth 3 Encoders 376
26 Booth Muxes 378
EX2
Rounding Constant With No Overflow or Dividend
4-2 CSA
4-2 CSA
4-2 CSA
4-2 CSA
Binary Tree 380
Rounding Constant With Overflow or Dividend
2
s,c
3,2 CSA (No Overflow) 382
3,2 CSA (With Overflow) 384
2
2
388
EX3
152b CPA 386
SB Logic
152b CPA
152b CPA 390
MSBs
To Div/Sqrt Round Logic
To Div/Sqrt Round Logic
MSBs
Below precision fixup 392
Below precision fixup 394
EX4
Round Control
Q, Q-1, Q+1
Rounded Mul, Ni, Di, Ti 396
Rounded Div/Sqrt 398
Ri 400
To Local Bypassing
Rounded Result Selection 402
To Local Bypassing

FIG. 5

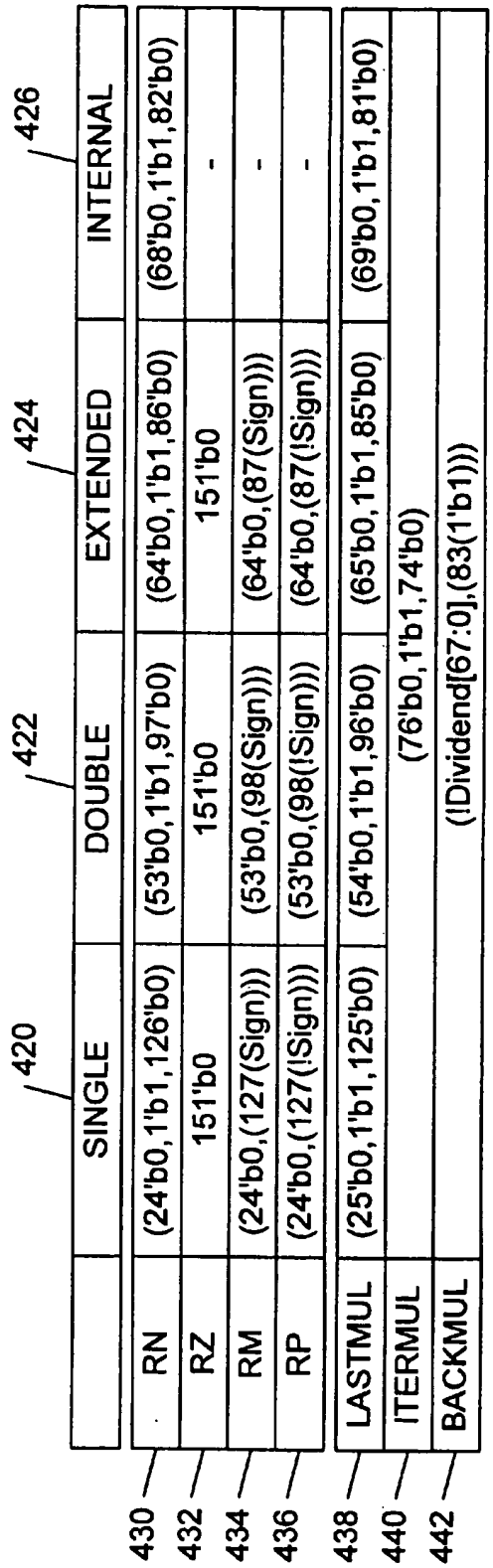

420
422
424
426
SINGLE
DOUBLE
EXTENDED
INTERNAL
RN
(24'b0,1'b1,126'b0)
(53'b0,1'b1,97'b0)
(64'b0,1'b1,86'b0)
(68'b0,1'b1,82'b0)
RZ
151'b0
151'b0
151'b0
-
RM
(24'b0,(127(Sign)))
(53'b0,(98(Sign)))
(64'b0,(87(Sign)))
-
RP
(24'b0,(127(!Sign)))
(53'b0,(98(!Sign)))
(64'b0,(87(!Sign)))
-
LASTMUL
(25'b0,1'b1,125'b0)
(54'b0,1'b1,96'b0)
(65'b0,1'b1,85'b0)
(69'b0,1'b1,81'b0)
ITERMUL
(76'b0,1'b1,74'b0)
BACKMUL
(!Dividend[67:0],(83(1'b1)))
430
432
434
436
438
440
442

FIG. 6

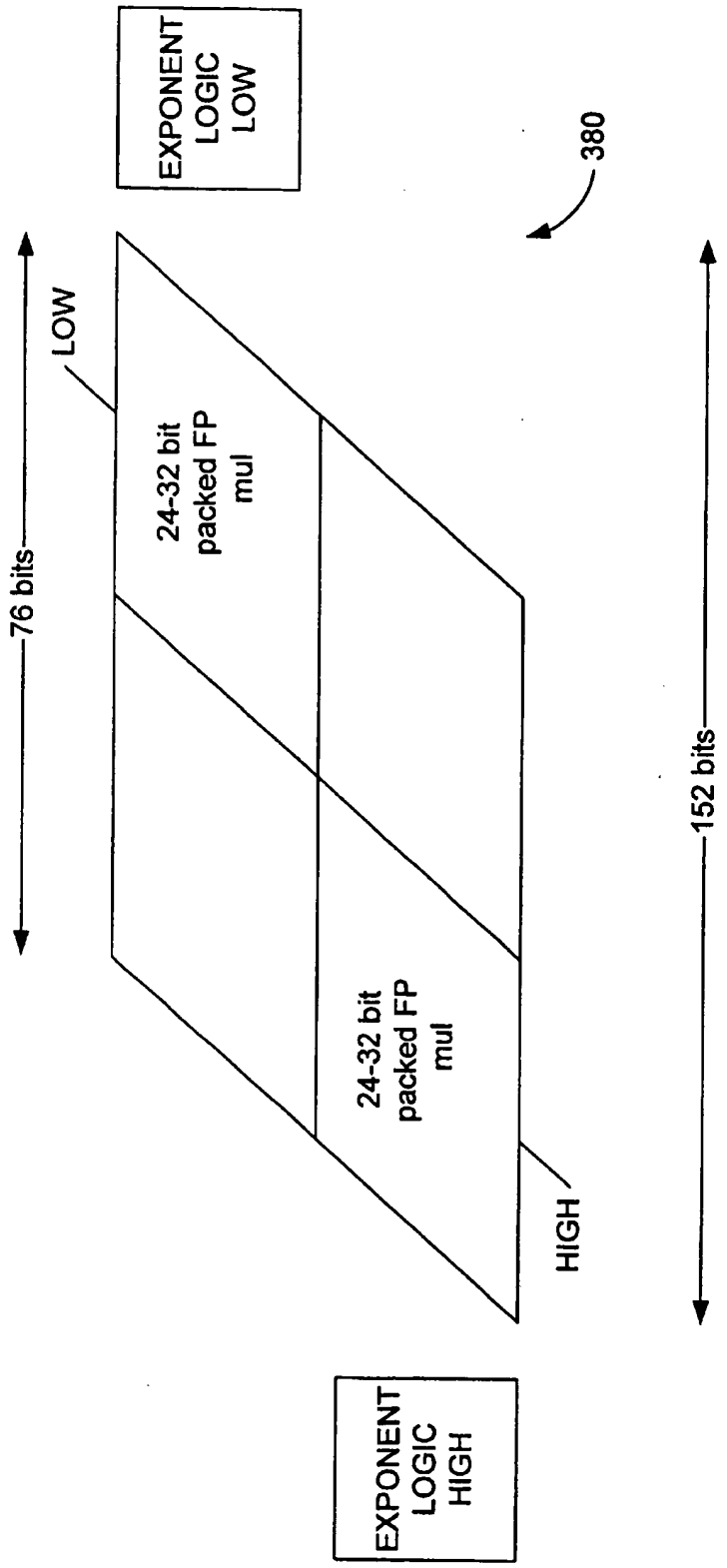

EXPONENT LOGIC LOW
380
LOW
76 bits
24-32 bit packed FP mul
24-32 bit packed FP mul
HIGH
152 bits
EXPONENT LOGIC HIGH

FIG. 7

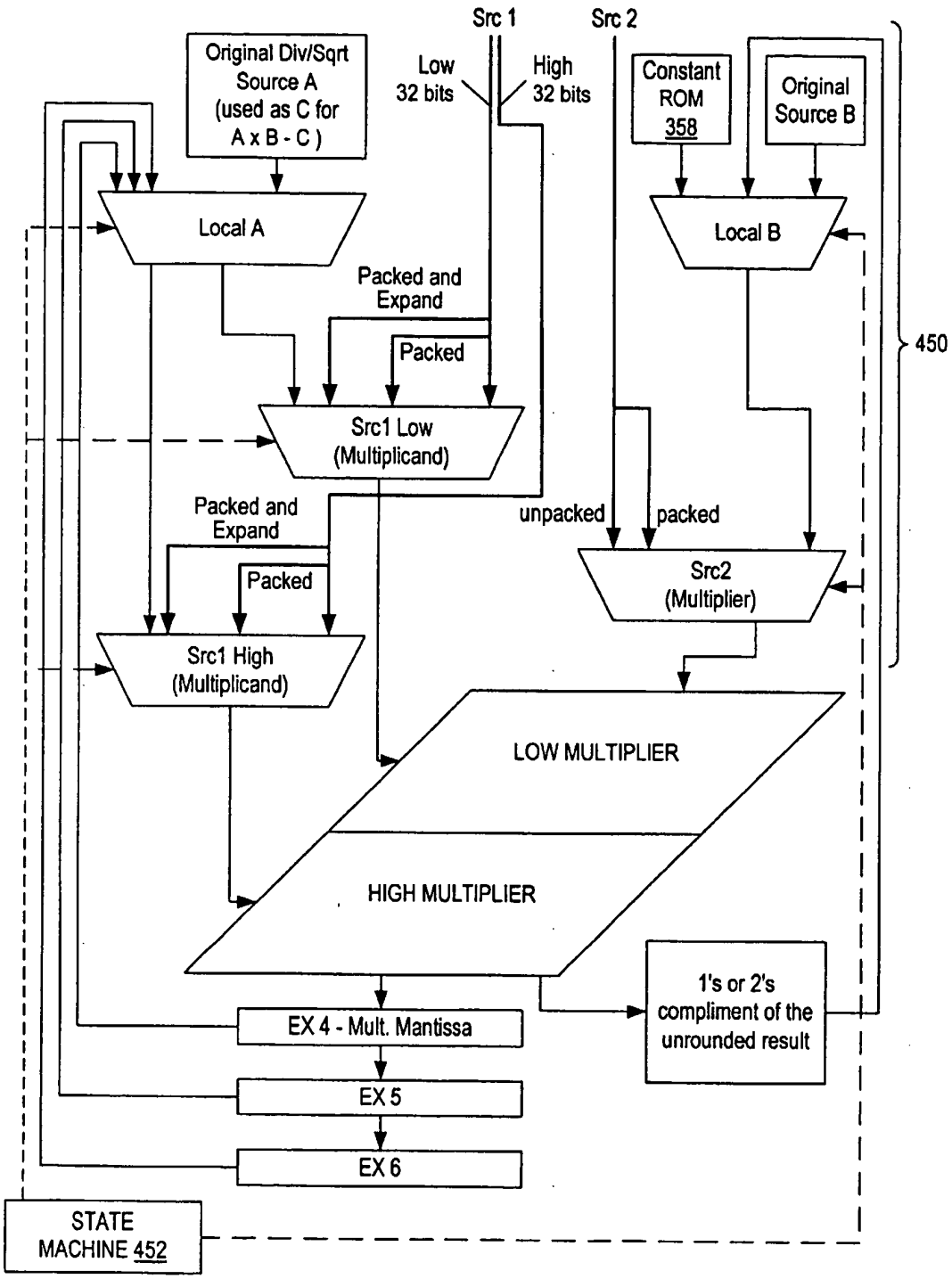

Src 1
Src 2
Original Div/Sqrt Source A (used as C for A x B - C)
Low 32 bits
High 32 bits
Constant ROM 358
Original Source B
Local A
Local B
Packed and Expand
Packed
Src1 Low (Multiplicand)
unpacked
packed
Packed and Expand
Packed
Src2 (Multiplier)
Src1 High (Multiplicand)
LOW MULTIPLIER
HIGH MULTIPLIER
1's or 2's compliment of the unrounded result
EX 4 - Mult. Mantissa
EX 5
EX 6
STATE MACHINE 452
450

FIG. 8

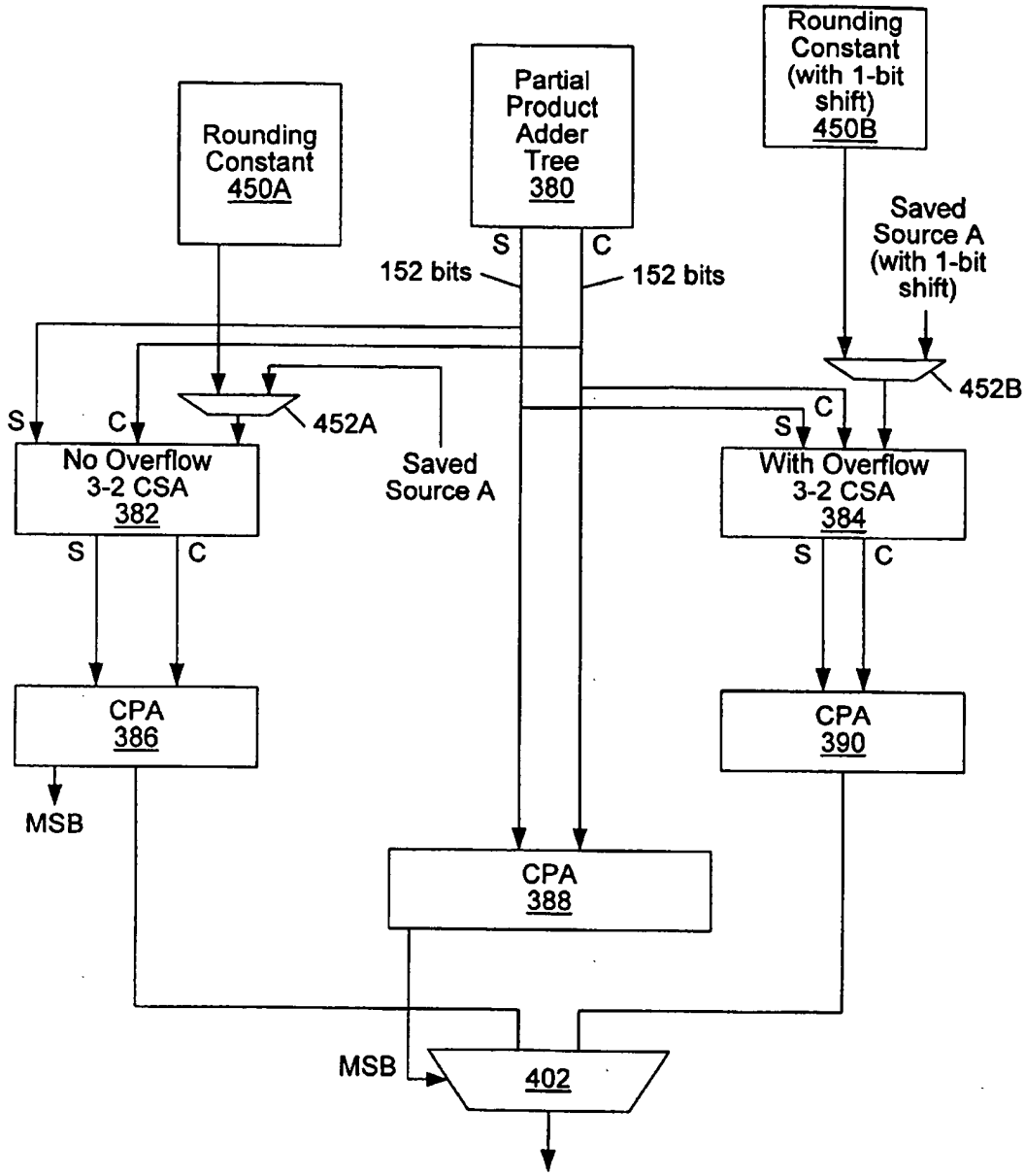

Rounding Constant 450A
Partial Product Adder Tree 380
Rounding Constant (with 1-bit shift) 450B
S
C
152 bits
152 bits
Saved Source A (with 1-bit shift)
452B
452A
S
C
C
S
No Overflow 3-2 CSA 382
Saved Source A
With Overflow 3-2 CSA 384
S
C
S
C
CPA 386
CPA 390
MSB
CPA 388
MSB
402

FIG. 9

Guard Bit	Rem-ainder	RN	RP (+/-)	RM (+/-)	RN
0	=0	trunc	trunc	trunc	trunc
0	-	trunc	trunc/dec	dec/trunc	dec
0	+	trunc	inc/trunc	trunc/inc	trunc
1	=0	RNE	inc/trunc	trunc/inc	trunc
1	-	trunc	inc/trunc	trunc/inc	trunc
1	+	inc	inc/trunc	trunc/inc	trunc

FIG. 10

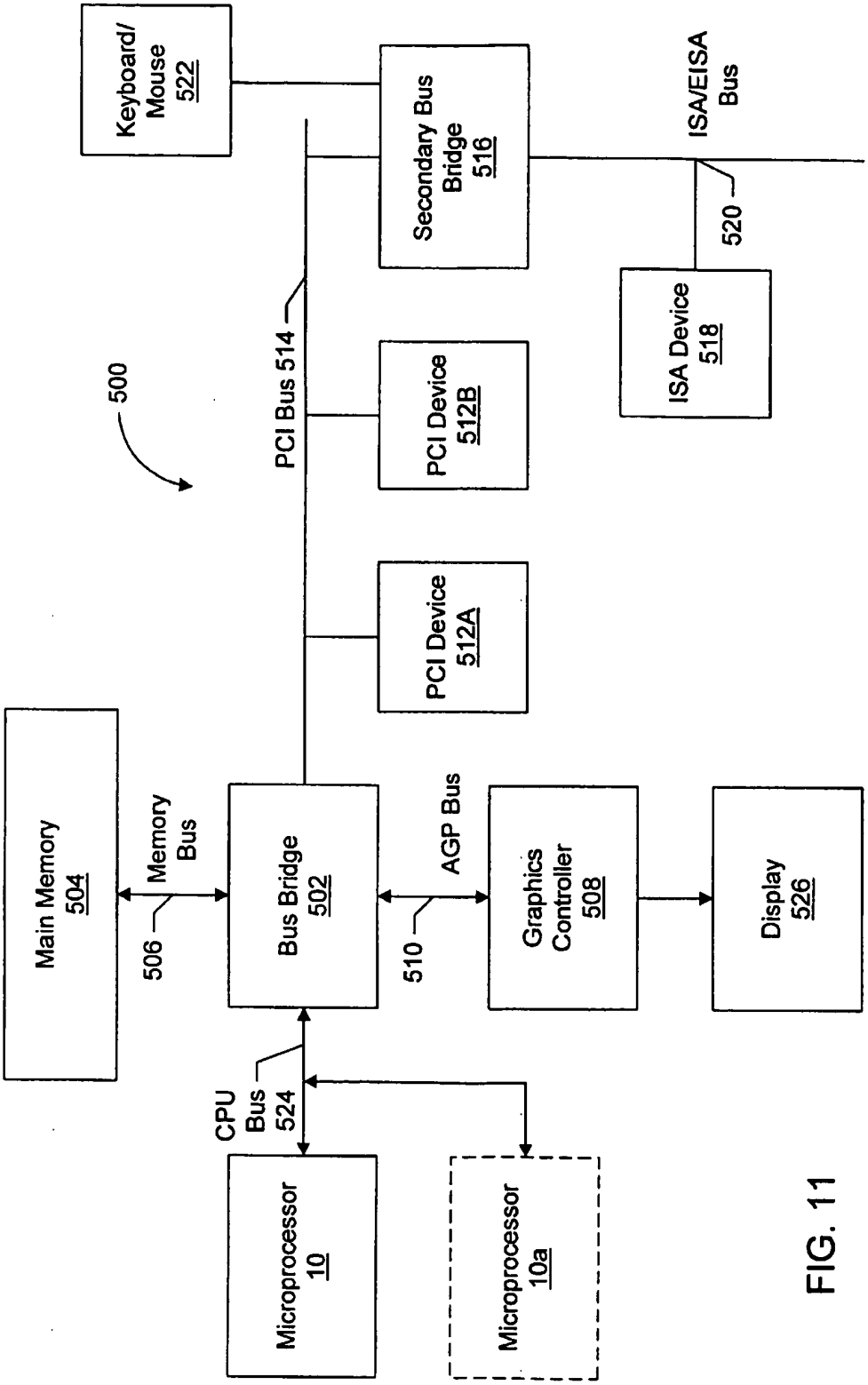

Keyboard/ Mouse 522
Secondary Bus Bridge 516
ISA/EISA Bus
520
ISA Device 518
500
PCI Bus 514
PCI Device 512B
PCI Device 512A
Main Memory 504
Memory Bus
506
Bus Bridge 502
AGP Bus
510
Graphics Controller 508
Display 526
CPU Bus 524
Microprocessor 10
Microprocessor 10a

FIG. 11

SHARED FP AND SIMD 3D MULTIPLIER

RELATED APPLICATIONS

This application is a continuation-in-part of the following: (i) U.S. application Ser. No. 09/014,454, filed on Jan. 1, 1998, now U.S. Pat. No. : 6,144,980; which is hereby incorporated by reference in its entirety; (ii) U.S. application Ser. No. 09/049,752, filed on Mar. 27, 1998, now U.S. Pat. No. : 6,269,384 which is hereby incorporated by reference in its entirety; and (iii) U.S. application Ser. No. 09/134,171, filed on Aug. 14, 1998, now U.S. Pat. No. : 6,223,198 which is hereby incorporated by reference in its entirety.

BACKGROUND OF THE INVENTION

1. Field of the Invention

This invention relates generally to the field of microprocessors and, more particularly, to multipliers that are usable to perform floating point calculations in microprocessors.

2. Description of the Related Art

Microprocessors are typically designed with a number of "execution units" that are each optimized to perform a particular set of functions or instructions. For example, one or more execution units within a microprocessor may be optimized to perform memory accesses, i.e., load and store operations. Other execution units may be optimized to perform general arithmetic and logic functions, e.g., shifts and compares. Many microprocessors also have specialized execution units configured to perform more complex floating-point arithmetic operations including multiplication and reciprocal operations. These specialized execution units typically comprise hardware that is optimized to perform one or more floating-point arithmetic functions.

Most microprocessors must support multiple data types. For example, x86 compatible microprocessors must execute instructions that are defined to operate upon an integer data type and instructions that are defined to operate upon floating-point data types. Floating-point data can represent numbers within a much larger range than integer data. For example, a 32-bit signed integer can represent the integers between -23^{31} and $2^{31}-1$ (using two's complement format). In contrast, a 32-bit ("single precision") floating-point number as defined by the Institute of Electrical and Electronic Engineers (IEEE) Standard 754 has a range (in normalized format) from 2^{-126} to $2^{127}\times(2-2^{-23})$ in both positive and negative numbers.

Turning now to FIG. **1A**, an exemplary format for an 8-bit integer **100** is shown. As illustrated in the figure, negative integers are represented using the two's complement format **104**. To negate an integer, all bits are inverted to obtain the one's complement format **102**. A constant of one is then added to the least significant bit (LSB).

Turning now to FIG. **1B**, an exemplary format for a 32-bit (single precision) floating-point number **106** is shown. A floating-point number is represented by a significand, an exponent and a sign bit. The base for the floating-point number is raised to the power of the exponent and multiplied by the significand to arrive at the number represented. In microprocessors, base **2** is typically used. The significand comprises a number of bits used to represent the most significant digits of the number. Typically, the significand comprises one bit to the left of the radix point and the remaining bits to the right of the radix point. In order to save space, the bit to the left of the radix point, known as the integer bit, is not explicitly stored. Instead, it is implied in the format of the number. Additional information regarding floating-point numbers and operations performed thereon may be obtained in IEEE Standard 754 (IEEE-754). Unlike the integer representation, two's complement format is not typically used in the floating-point representation. Instead, sign and magnitude form are used. Thus, only the sign bit is changed when converting from positive value **106** to negative value **108**.

In the x86 architecture, the floating point format supports a number of special cases. These special cases may appear in one or more operands or one or more results for a particular instruction. FIG. **2** illustrates the sign, exponent, and significand formats of special and exceptional cases that are included in the IEEE-754 floating-point standard. The special and exceptional cases shown in FIG. **2** include a zero value, an infinity value, NaN (not-a-number) values, and a denormal value. An 'x' in FIG. **2** represents a value that can be either one or zero. NaN values may include a QNaN (quiet not-a-number) value and a SNaN (signaling not-a-number) value as defined by a particular architecture. The numbers depicted in FIG. **2** are shown in base **2** format as indicated by the subscript **2** following each number. As shown, a number with all zeros in its exponent and significand represents a zero value in the IEEE-754 floating-point standard. A number with all ones in its exponent, a one in the most significant bit of its significand, and zeros in the remaining bits of its significant represents an infinity value. The remaining special and exceptional cases are depicted similarly.

Given the substantial differences in floating point and integer formats, microprocessor designers have typically used two sets of execution units, i.e., one set optimized to perform arithmetic on integer instructions and one set optimized to perform arithmetic on floating point instructions. Unfortunately, this approach has some potential drawbacks. Die space on a microprocessor is a relatively scarce commodity, and the die space required to implement complex execution units such as multipliers is significant. Thus, duplicating multipliers for both integer and floating point formats consumes precious real estate that could be used to implement additional functionality.

The recent addition of three-dimensional graphics instructions (e.g., AMD's 3DNow™ instructions) to the standard x86 instruction has further complicated matters by increasing the performance demands on the microprocessor's arithmatic execution units (and multiplier execution units in particular). As those skilled in the art will appreciate, 3DNow™ instructions are so-called SIME (single instruction, multiple data) instructions that have operands that include multiple floating point values packed together.

As a result, a method for executing arithmetic instructions with different instruction and data formats is needed. In particular, a method for executing multiply instructions having different data types without dramatically increasing the die space used is desired.

SUMMARY

The problems outlined above are in large part solved by the multiplier and method for performing multiplication described herein. In one embodiment, a single multiplier may be configured to perform scalar floating point multiplication and packed floating point multiplication, e.g., single-instruction multiple-data (SIMD) multiplication.

The multiplier may include selection logic that is configured to select a multiplier operand and a multiplicand operand from among a plurality of different potential sources, wherein the potential sources may include one or

more of the following: a floating point operand, a packed floating point operand, or the result of a previous iterative multiplication instruction. For example, the multiplier may perform scalar 90-bit format floating point multiplication (i.e., single, double, extended, or internal precision), packed 90-bit format (i.e., 2 packed 32-bit floating point values); or the results of a previous multiplication instruction (i.e., via an internal bypass mechanism for instructions such as iterative multiplication operations). For example, some instructions such as reciprocal instructions (used to perform division) and square root instructions may be implemented using iterative algorithms that perform a number of different multiplication operations before obtaining the correct result. In these situations it may particularly advantageous for a multiplier to have the capability of internally bypassing the results of a previous multiplication operation directly to the selection logic for another multiplication operation.

In some embodiments, square root and divide instructions may be translated into a series of special multiplication opcodes, wherein each multiplication opcode is configured to perform a particular function in addition to standard multiplication. For example, one of the iterative special multiplication instructions may be a "BACKMUL" instruction that accepts three source operands, A, B, and Q, and calculates the value B·Q−A.

In some embodiments, the multiplier may be further configured to detect divide operations having a multiplier that is exactly a power of two. The multiplier may be configured to execute these divide operations without proceeding through the entire iterative division process. Instead, the multiplier may be configured to shift the exponent and round the significand to the appropriate precision without significant additional hardware.

In some embodiments, the multiplier may be further configured to perform independent multiplication instructions during the idle clock cycles that may occur during complex iterative instructions such as square root and non-power of two divides.

BRIEF DESCRIPTION OF THE DRAWINGS

Other objects and advantages of the invention will become apparent upon reading the following detailed description and upon reference to the accompanying drawings in which:

FIG. **1A** is a diagram of an exemplary integer data format using two's complement representation.

FIG. **1B** is a diagram of an exemplary floating-point data format.

FIG. **2** is a table listing special cases for a defined floating-point data format.

FIG. **3** is a block diagram of one embodiment of an exemplary microprocessor.

FIG. **4** is a block diagram of one embodiment of a floating-point unit from the exemplary microprocessor of FIG. **3**.

FIG. **5** is a block diagram depicting portions of one embodiment of the multiplier from FIG. **4**.

FIG. **6** is a table illustrating rounding constants for one embodiment of the multiplier from FIG. **5**.

FIG. **7** is a block diagram illustrating generally one embodiment of the binary adder tree from the multiplier of FIG. **5**.

FIG. **8** is a block diagram illustrating more details of one embodiment of the multiplier from FIG. **7**.

FIG. **9** is a block diagram illustrating more details of one embodiment of the selection logic of the multiplier from FIG. **5**.

FIG. **10** is a table of rounding actions for one embodiment of the multiplier from FIG. **5**.

FIG. **11** is a block diagram of one embodiment of a computer system configured to utilize the microprocessor of FIG. **1**.

While the invention is susceptible to various modifications and alternative forms, specific embodiments thereof are shown by way of example in the drawings and will herein be described in detail. It should be understood, however, that the drawings and detailed description thereto are not intended to limit the invention to the particular form disclosed, but on the contrary, the intention is to cover all modifications, equivalents, and alternatives falling within the spirit and scope of the present invention as defined by the appended claims.

DETAILED DESCRIPTION OF SEVERAL EMBODIMENTS

Turning now to FIG. **3**, a block diagram of one embodiment of a microprocessor **10** is shown. Microprocessor **10** includes a prefetch/predecode unit **12**, a branch prediction unit **14**, an instruction cache **16**, an instruction alignment unit **18**, a plurality of decode units **20A–20C**, a plurality of reservation stations **22A–22C**, a plurality of functional units **24A–24C**, a load/store unit **26**, a data cache **28**, a register file **30**, a reorder buffer **32**, an MROM unit **34**, and a floating-point unit (FPU) **36**, which in turn comprises multiplier **50**. Note that elements referred to herein with a particular reference number followed by a letter may be collectively referred to by the reference number alone. For example, decode units **20A–20C** may be collectively referred to as decode units **20**.

Prefetch/predecode unit **12** is coupled to receive instructions from a main memory subsystem (not shown), and is further coupled to instruction cache **16** and branch prediction unit **14**. Similarly, branch prediction unit **14** is coupled to instruction cache **16**. Still further, branch prediction unit **14** is coupled to decode units **20** and functional units **24**. Instruction cache **16** is further coupled to MROM unit **34** and instruction alignment unit **18**. Instruction alignment unit **18**, which comprises an early decode unit (EDU) **44**, is in turn coupled to decode units **20**. Each decode unit **20A–20C** is coupled to load/store unit **26** and to respective reservation stations **22A–22C**. Reservation stations **22A–22C** are further coupled to respective functional units **24A–24C**. Additionally, decode units **20** and reservation stations **22** are coupled to register file **30** and reorder buffer **32**. Functional units **24** are coupled to load/store unit **26**, register file **30**, and reorder buffer **32** as well. Data cache **28** is coupled to load/store unit **26** and to the main memory subsystem. MROM unit **34**, which also comprises an early decode unit (EDU) **42** is coupled to decode units **20** and FPU **36**. Finally, FPU **36** is coupled to load/store unit **26** and reorder buffer **32**.

Instruction cache **16** is a high speed cache memory provided to store instructions. Instructions are fetched from instruction cache **16** and dispatched to decode units **20**. In one embodiment, instruction cache **16** is configured to store up to 64 kilobytes of instructions in a 2-way set associative structure having 64-byte lines (a byte comprises 8 binary bits). It is noted that instruction cache **16** may be implemented in a fully-associative, set-associative, or direct-mapped configuration.

Instructions are fetched from main memory and stored into instruction cache **16** by prefetch/predecode unit **12**. Instructions may be prefetched prior to the request thereof in

accordance with a prefetch scheme. A variety of prefetch schemes may be employed by prefetch/predecode unit **12**. As prefetch/predecode unit **12** transfers instructions from main memory to instruction cache **16**, prefetch/predecode unit **12** generates three predecode bits for each byte of the instructions: a start bit, an end bit, and a functional bit. The predecode bits form tags indicative of the boundaries of each instruction. The predecode tags may also convey additional information such as whether a given instruction may be decoded directly by decode units **20** or whether the instruction is executed by invoking a microcode procedure controlled by MROM unit **34**, as will be described in greater detail below. Still further, prefetch/predecode unit **12** may be configured to detect branch instructions and to store branch prediction information corresponding to the branch instructions into branch prediction unit **14**.

One encoding of the predecode tags for an embodiment of microprocessor **10** employing a variable byte length instruction set will next be described. A variable byte length instruction set is an instruction set in which different instructions may occupy differing numbers of bytes. An exemplary variable byte length instruction set employed by one embodiment of microprocessor **10** is the x86 instruction set.

In the exemplary encoding, if a given byte is the first byte of an instruction, the start bit for that byte is set. If the byte is the last byte of an instruction, the end bit for that byte is set. Instructions which may be directly decoded by decode units **20** are referred to as "fast path" instructions. The remaining x86 instructions are referred to as MROM instructions, according to one embodiment. For fast path instructions, the functional bit is set for each prefix byte included in the instruction, and cleared for other bytes. Alternatively, for MROM instructions, the functional bit is cleared for each prefix byte and set for other bytes. The type of instruction may be determined by examining the functional bit corresponding to the end byte. If that functional bit is clear, the instruction is a fast path instruction. Conversely, if that functional bit is set, the instruction is an MROM instruction. The opcode of an instruction may thereby be located within an instruction which may be directly decoded by decode units **20** as the byte associated with the first clear functional bit in the instruction. For example, a fast path instruction including two prefix bytes, a Mod RIM byte, and an immediate byte would have start, end, and functional bits as follows:

Start bits	10000
End bits	00001
Functional bits	11000

According to one particular embodiment, early identifying that an instruction includes a scale-index-base (SIB) byte is advantageous for MROM unit **34**. For such an embodiment, if an instruction includes at least two bytes after the opcode byte, the functional bit for the Mod R/M byte indicates the presence of an SIB byte. If the functional bit for the Mod R/M byte is set, then an SIB byte is present. Alternatively, if the functional bit for the Mod R/M byte is clear, then an SIB byte is not present.

MROM instructions are instructions which are determined to be too complex for decode by decode units **20**. MROM instructions are executed by invoking MROM unit **34**. More specifically, when an MROM instruction is encountered, MROM unit **34** parses and issues the instruction into a subset of defined fast path instructions to effectuate the desired operation. MROM unit **34** dispatches the subset of fast path instructions to decode units **20**.

Microprocessor **10** employs branch prediction in order to speculatively fetch instructions subsequent to conditional branch instructions. Branch prediction unit **14** is included to perform branch prediction operations. In one embodiment, up to two branch target addresses are stored with respect to each 16 byte portion of each cache line in instruction cache **16**. Prefetch/predecode unit **12** determines initial branch targets when a particular line is predecoded. Subsequent updates to the branch targets corresponding to a cache line may occur due to the execution of instructions within the cache line. Instruction cache **16** provides an indication of the instruction address being fetched, so that branch prediction unit **14** may determine which branch target addresses to select for forming a branch prediction. Decode units **20** and functional units **24** provide update information to branch prediction unit **14**. Because branch prediction unit **14** stores two targets per 16 byte portion of the cache line, some branch instructions within the line may not be stored in branch prediction unit **14**. Decode units **20** detect branch instructions which were not predicted by branch prediction unit **14**. Functional units **24** execute the branch instructions and determine if the predicted branch direction is incorrect. The branch direction may be "taken", in which subsequent instructions are fetched from the target address of the branch instruction. Conversely, the branch direction may be "not taken", in which case subsequent instructions are fetched from memory locations consecutive to the branch instruction. When a mispredicted branch instruction is detected, instructions subsequent to the mispredicted branch are discarded from the various units of microprocessor **10**. A variety of suitable branch prediction algorithms may be employed by branch prediction unit **14**.

Instructions fetched from instruction cache **16** are conveyed to instruction alignment unit **18** and MROM unit **34**. As instructions are fetched from instruction cache **16**, the corresponding predecode data is scanned to provide information to instruction alignment unit **18** and MROM unit **34** regarding the instructions being fetched. Instruction alignment unit **18** utilizes the scanning data to align an instruction to each of multiplexers **46A–C**. In one embodiment, instruction alignment unit **18** aligns instructions from three sets of eight instruction bytes to decode units **20**. Similarly, MROM unit **34** is configured to output up to three aligned instructions to multiplexers **46A–C**. Note, both instruction alignment unit **18** and MROM unit **34** may each have an early decode unit (EDC) **42** and **44**. These units may perform the first steps of the decoding process, e.g., identifying the operand specifiers for each instruction.

Each multiplexer **46A–C** is configured to receive a partially decoded instruction (and corresponding decode and predecode information) from instruction alignment unit **18** and MROM unit **34**. Each multiplexer **46A–C** is configured to select up to one instruction from either instruction alignment unit **18** or MROM unit **34** during each clock cycle. The selected instructions are routed to decode units **20A–C** (integer instructions), and FPU **36** (x86 floating-point, MMX, and 3DX instructions). In one embodiment of microprocessor **10**, up to three floating-point instructions per clock cycle may be conveyed to floating-point unit **36**. As note above, the instructions may come from MROM unit **34** (microcode instructions) or instruction alignment unit **18** (fast path instructions). Decode units **20** are configured to complete decoding instructions received from multiplexers **46A–C**. Register operand information is detected and routed to register file **30** and reorder buffer **32**. Additionally, if the

instructions require one or more memory operations to be performed, decode units **20** dispatch the memory operations to load/store unit **26**. Each instruction is decoded into a set of control values for functional units **24**, and these control values are dispatched to reservation stations **22** along with operand address information and displacement or immediate data which may be included with the instruction.

Microprocessor **10** supports out of order execution, and thus employs reorder buffer **32** to keep track of the original program sequence for register read and write operations, to implement register renaming, to allow for speculative instruction execution and branch misprediction recovery, and to facilitate precise exceptions. A temporary storage location within reorder buffer **32** is reserved upon decode of an instruction that involves the update of a register to thereby store speculative register states. If a branch prediction is incorrect, the results of speculatively-executed instructions along the mispredicted path can be invalidated in the buffer before they are written to register file **30**. Similarly, if a particular instruction causes an exception, instructions subsequent to the particular instruction may be discarded. In this manner, exceptions are "precise" (i.e., instructions subsequent to the particular instruction causing the exception are not retired prior to the exception). Stated another way, while some instructions following the exception-causing instruction may have been executed before the exception, their results have not been committed to the microprocessor's architectural state. It is noted that a particular instruction is speculatively executed if it is executed prior to instructions which precede the particular instruction in program order. Preceding instructions may be a branch instruction or an exception-causing instruction, in which case the speculative results may be discarded by reorder buffer **32**.

The instruction control values and immediate or displacement data provided at the outputs of decode units **20** are routed directly to respective reservation stations **22**. In one embodiment, each reservation station **22** is capable of holding instruction information (i.e., instruction control values as well as operand values, operand tags and/or immediate data) for up to six pending instructions awaiting issue to the corresponding functional unit. It is noted that for the embodiment of FIG. **3**, each reservation station **22** is associated with a dedicated functional unit **24**. Accordingly, three dedicated "issue positions" are formed by reservation stations **22** and functional units **24**. In other words, issue position **0** is formed by reservation station **22A** and functional unit **24A**. Instructions aligned and dispatched to reservation station **22A** are executed by functional unit **24A**. Similarly, issue position **1** is formed by reservation station **22B** and functional unit **24B**; and issue position **2** is formed by reservation station **22C** and functional unit **24C**.

Upon decode of a particular instruction, if a required operand is a register location, register address information is routed to reorder buffer **32** and register file **30** simultaneously. Those of skill in the art will appreciate that the x86 register file includes eight 32 bit real registers (i.e., typically referred to as EAX, EBX, ECX, EDX, EBP, ESI, EDI and ESP). In embodiments of microprocessor **10** which employ the x86 microprocessor architecture, register file **30** comprises storage locations for each of the 32 bit real registers. Additional storage locations may be included within register file **30** for use by MROM unit **34**. Reorder buffer **32** contains temporary storage locations for results which change the contents of these registers to thereby allow out of order execution. A temporary storage location of reorder buffer **32** is reserved for each instruction which, upon decode, is determined to modify the contents of one of the real registers. Therefore, at various points during execution of a particular program, reorder buffer **32** may have one or more locations which contain the speculatively executed contents of a given register. If following decode of a given instruction it is determined that reorder buffer **32** has a previous location or locations assigned to a register used as an operand in the given instruction, the reorder buffer **32** forwards to the corresponding reservation station either: 1) the value in the most recently assigned location, or 2) a tag for the most recently assigned location if the value has not yet been produced by the functional unit that will eventually execute the previous instruction. If reorder buffer **32** has a location reserved for a given register, the operand value (or reorder buffer tag) is provided from reorder buffer **32** rather than from register file **30**. If there is no location reserved for a required register in reorder buffer **32**, the value is taken directly from register file **30**. If the operand corresponds to a memory location, the operand value is provided to the reservation station through load/store unit **26**.

In one particular embodiment, reorder buffer **32** is configured to store and manipulate concurrently decoded instructions as a unit. This configuration will be referred to herein as "line-oriented". By manipulating several instructions together, the hardware employed within reorder buffer **32** may be simplified. For example, a line-oriented reorder buffer included in the present embodiment allocates storage sufficient for instruction information pertaining to three instructions (one from each decode unit **20**) whenever one or more instructions are dispatched by decode units **20**. By contrast, a variable amount of storage is allocated in conventional reorder buffers, dependent upon the number of instructions actually dispatched. A comparatively larger number of logic gates may be required to allocate the variable amount of storage. When each of the concurrently decoded instructions has executed, the instruction results are stored into register file **30** simultaneously. The storage is then free for allocation to another set of concurrently decoded instructions. Additionally, the amount of control logic circuitry employed per instruction is reduced because the control logic is amortized over several concurrently decoded instructions. A reorder buffer tag identifying a particular instruction may be divided into two fields: a line tag and an offset tag. The line tag identifies the set of concurrently decoded instructions including the particular instruction, and the offset tag identifies which instruction within the set corresponds to the particular instruction. It is noted that storing instruction results into register file **30** and freeing the corresponding storage is referred to as "retiring" the instructions. It is further noted that any reorder buffer configuration may be employed in various embodiments of microprocessor **10**.

As noted earlier, reservation stations **22** store instructions until the instructions are executed by the corresponding functional unit **24**. An instruction is selected for execution if both: (i) the operands of the instruction have been provided; and (ii) the operands have not yet been provided for instructions which are within the same reservation station **22A–22C** and which are prior to the instruction in program order. It is noted that when an instruction is executed by one of the functional units **24**, the result of that instruction is passed directly to any reservation stations **22** that are waiting for that result at the same time the result is passed to update reorder buffer **32** (this technique is commonly referred to as "result forwarding"). An instruction may be selected for execution and passed to a functional unit **24A–24C** during the clock cycle that the associated result is forwarded.

Reservation stations **22** route the forwarded result to the functional unit **24** in this case.

In one embodiment, each of the functional units **24** is configured to perform integer arithmetic operations of addition and subtraction, as well as shifts, rotates, logical operations, and branch operations. The operations are performed in response to the control values decoded for a particular instruction by decode units **20**. Additionally, functional units **24** may be configured to perform address generation for load and store memory operations performed by load/store unit **26**. In one particular embodiment, each functional unit **24** may comprise an execution unit and an independent address generation unit. Such functional units may perform an address generation for conveyance to load/store unit **26** in parallel with the execution of an integer or branch operation.

Each of the functional units **24** also provides information regarding the execution of conditional branch instructions to the branch prediction unit **14**. If a branch prediction was incorrect, branch prediction unit **14** flushes instructions subsequent to the mispredicted branch that have entered the instruction processing pipeline, and causes fetch of the required instructions from instruction cache **16** or main memory. It is noted that in such situations, results of instructions in the original program sequence which occur after the mispredicted branch instruction are discarded, including those which were speculatively executed and temporarily stored in load/store unit **26** and reorder buffer **32**.

Floating Point Unit

Results produced by functional units **24** are sent to reorder buffer **32** if a register value is being updated, and to load/store unit **26** if the contents of a memory location are changed. If the result is to be stored in a register, reorder buffer **32** stores the result in the location reserved for the value of the register when the instruction was decoded. A plurality of result buses **38** are included for forwarding of results from functional units **24** and load/store unit **26**. Result buses **38** convey the result generated, as well as the reorder buffer tag identifying the instruction being executed.

Load/store unit **26** provides an interface between functional units **24** and data cache **28**. In one embodiment, load/store unit **26** is configured with a pre-cache load/store buffer having twelve storage locations for data and address information for pending loads or stores and a post-cache load/store buffer having **32** entries. Decode units **20** arbitrate for access to the load/store unit **26**. When the buffer is full, a decode unit must wait until load/store unit **26** has room for the pending load or store request information. Load/store unit **26** also performs dependency checking for load memory operations against pending store memory operations to ensure that data coherency is maintained. A memory operation is a transfer of data between microprocessor and the main memory subsystem. Memory operations may be the result of an instruction which utilizes an operand stored in memory, or may be the result of a load/store instruction which causes the data transfer but no other operation. Additionally, load/store unit **26** may include a special register storage for special registers such as the segment registers and other registers related to the address translation mechanism defined by the x86 microprocessor architecture.

Data cache **28** is a high speed cache memory provided to temporarily store data being transferred between load/store unit **26** and the main memory subsystem. In one embodiment, data cache **28** has a capacity of storing up to sixty-four kilobytes of data in a two way set associative structure. It is understood that data cache **28** may be implemented in a variety of specific memory configurations, including a set associative configuration.

Turning now to FIG. **4**, details of one embodiment of FPU **36** are shown. Other embodiments are possible and contemplated. FPU **36** is a high performance out-of-order execution unit capable of accepting up to three new instructions per clock cycle. The three instructions may be any combination of x86 floating-point instructions, MMX instructions, or 3DX instructions. MMX and 3DX instructions are extensions to the standard x86 instruction set. One example of a 3DX instruction set extension is the 3DNow!™ extension from Advanced Micro Devices, Inc. MMX instructions are geared toward multimedia and two-dimensional graphic applications, while 3DX instructions are optimized for performing three-dimensional graphic manipulations such as rendering and texture mapping. Many 3DX instructions are vectored instructions that perform the same operation on a number of independent pairs of operands.

As the figure illustrates, this embodiment of FPU **36** comprises the following components: a rename-**1** unit **310**, a rename-2 unit **312**, a scheduler **314**, a retire queue **316**, a register file **318**, a load/add execution pipeline **320**, a load/multiply execution pipeline **322**, a load/store execution pipeline **326**, a skid buffer **334**, a convert and classify unit **336** and a load mapper **338**. Rename-**1** unit **310** is coupled to rename-2 unit **312** and is configured to receive a speculative top of stack (TOS) tag and tag word **352**. Rename-2 unit **312** is coupled to future file tag array **328**, architectural tag array **330**, retire queue **316**, skid buffer **334**, scheduler **314**, and load mapper **338**. Convert and classify unit **336** is also coupled to load mapper **338**, which in turn is coupled to execution and pipeline control unit **340** along with instruction status register file **342** and scheduler **314**. Register file **318** receives inputs from convert and classify unit **336**, load mapper **338** and scheduler **314**, and outputs data to source operand bus **348**. Source operand bus **348** is in turn coupled to execution pipelines **320**, **322**, and **326**. Finally, execution pipelines **320**, **322**, and **326**, and floating-point status/control/tag words **344** are all coupled to result bus **346**. While floating-point status/control/tag words **344** and speculative top of stack and tag word **352** are shown separately in the figure for explanatory purposes, these tags may be stored together with future file tags **328** and architectural register tags **330** in a single storage location, e.g., within register file **318**, execution and pipeline control unit **340**, or retire queue **316**.

Rename-**1** unit **310** receives up to three instructions per clock cycle. As previously noted, these may be any combination of floating-point, MMX, or 3DX instructions. Rename-**1** unit **310** converts stack-relative registers into absolute register numbers. For instructions with memory operands, e.g., FLD instructions (floating-point load), a stack-relative reference (e.g., the destination ST(7)) is mapped to an absolute register number. Furthermore, in some embodiments load-execute instructions such as FADD [mem] also need to source operands converted from top-of-stack relative addressing to absolute addressing. Thus, in some embodiments x87 type instructions (i.e., floating point instructions) go through the stack to absolute register translation process, while MMX and 3DNow! instructions do not. The x86 instruction set and architecture defines eight floating-point registers that are accessed in a stack-like manner (i.e., relative to a top-of-stack pointer). Rename-**1** unit **310** also assigns each instruction to one of three execution pipelines, either load/store execution pipeline

326, load/add execution pipeline 320, or load/multiply execution pipeline 322 and, if necessary, converts each instruction to an internal format.

Rename-2 unit 312 performs true register renaming. Upon receiving the instructions from rename-1 unit 310, rename-2 unit 312 reads three register tags from a "free list" of the available registers stored within retire queue 316. Once the registers have been read, rename-2 unit 312 assigns one to the destination register of each instruction. To rename the source registers, rename-2 unit 312 indexes tag future file 328 using the absolute register number for each source register. Tag future file 328 stores tags that identify which registers store the current speculative future state of each of the sixteen architectural registers in FPU 36. Similarly, architectural register tag file 330 stores tags which identify which registers within register file 318 store the current architectural (non-speculative) state of FPU 36. Note, of the sixteen registers that define FPU 36's state (architectural or speculative), eight are architectural registers (i.e., floating-point stack or MMX registers) and eight are micro-architectural registers (i.e., registers that store internal state information that is not generally accessible to the programmer). The old destination register tags are then read from the tag future file 328 and written to the tail of the free list. Finally, tag future file 328 is updated by storing tags for the new destination registers.

Memory source operands may be handled by assigning them the same register tag as the destination register. This is because load data will be converted and directly written into the destination register when it is received from load/store unit 26. In the case of an FLD instruction, no further processing is required (except in certain exceptional cases), although the FLD instruction is still assigned to an execution pipeline for the purpose of handling exceptions and signaling completion to reorder buffer 32.

Once the three instructions have passed through rename-1 unit 310 and rename-2 unit 312, the instructions are represented in a three operand format (i.e., first source operand, second source operand, and destination operand). While the first source operand is always a register operand, a bit in the opcode may be used to indicate whether the second operand is a register operand or a memory operand.

From rename-2 unit 312 the instructions are passed to scheduler 314, where the three instructions are allocated a "line" of storage. If scheduler 314 is full, the instructions may be stored in skid buffer 334 until such time as there is room within scheduler 314. After receiving the three instructions, scheduler 314 snoops result bus 346 and source operand bus 348. Scheduler 314 may also snoop load data bus. Concurrently with allocating the line of storage and snooping, retire queue 316 allocates one entry for each instruction. The entries store the destination register tags, the absolute destination register number, and the old destination register tags. Additional information may also be included, e.g., information that may be needed to update the architectural state at retire time.

On the cycle following their entry into scheduler 314, the instructions are available for scheduling. Scheduler 314 examines all of the stored instructions and issues the oldest instructions which meet the following criteria: (1) the execution pipeline to which the instruction has been assigned is available, (2) the result bus for that execution pipeline will be available on the clock cycle in which the instruction will complete (this is dependent upon the latency of the particular instruction), and (3) the instruction's source registers and or memory operands are available. In this embodiment, scheduler 314 may schedule up to three instructions per clock cycle. Each of the three execution pipelines 320, 322, and 326 may accept a new instruction every clock cycle. Note other embodiments capable of scheduling four or more instructions are also possible and contemplated.

Once all three entries in a line are scheduled, that line is free to be compacted out of scheduler 314. When the instructions are conveyed from scheduler 314 to their respective instruction execution pipeline, their source operands are read. In some cases, the source data will come from a register, while in other cases the source data will come from a "bypass". A bypass refers to the practice of result forwarding or superforwarding. Result forwarding involves conveying the results from a recently executed instruction directly to other instructions that depend upon that result. Result forwarding allows the result to be used in subsequent instructions without having to wait for the result to be stored in a register and having to read the result from the register. Result superforwarding will be described in more detail below.

Each execution pipeline 320, 322, and 326 may be configured as a four-stage pipeline. In the first stage of each pipeline, the result buses are read and the input data is taken from either the source operand bus (if the data is within register file 318) or the result bus (if a bypass is being performed). Once the source data has been received, each execution pipeline may begin performing the appropriate computation.

In the embodiment shown in the figure, execution pipeline 320 is configured to perform load and addition operations, execution pipeline 322 is configured to perform load and multiplication operations, and execution pipeline 326 is configured to perform load and store operations. Both execution pipelines 320 and 322 may be configured to perform certain MMX instructions. Execution pipeline 322, which comprises multiplier 50, may also be configured to perform iterative calculations that involve multiplication, e.g., reciprocal operations, division operations, and square root operations, under the control of control unit 354, division/square root ROM 360, and, if a remainder is called for, remainder control unit 356. Constant ROM 358 is a read only memory configured to store a plurality of constants for load constant instructions such as FLDPI, for transcendental computation, for FPU 36 self-checking, and for certain special and exceptional results. Division/square root ROM 360 is a read only memory which stores constants used to determine initial values for division and square root computations and constants returned by certain 3DNow! instructions. Control unit 354 provides sequence information for division and square root functions. Note, in some embodiments control unit 354 may be part of execution and pipeline control unit 340.

In some cases, floating point instruction operands or floating point results generated by executing an instruction may be too small to fit within the operand or result's standard data format. These numbers are referred to as "denormals". While normalized floating-point values have a non-zero exponent and a one in the most significant bit of the significand, i.e., the bit directly to the left of the binary radix point (e.g., 1.001010 . . .), denormals are represented with a zero exponent and a zero in the most significant bit of the significand (e.g., 0.000101 . . .). Denormal load data is detected and tagged by convert and classify unit 336. Denormal results generated by during execution within execution pipelines 320, 322, and 326 are tagged when they are generated. Execution and pipeline control unit 340 detects the presence of the denormal tags and calls an

appropriate microcode routine from MROM **34** to handle the denormal data.

At the end of the final execution stage, the data is placed on result bus **346**. This makes the result data available for an instruction entering the first stage of one of the instruction execution pipelines during the next clock cycle. Once the data is on the result bus, it may also be written into register file **318**. Instead of being stored in register file **318**, store data is sent to the load/store unit **26**. In some cases, however, store data may be sent to both the load/store unit **26** and register file **318** (e.g., for floating point load control word instructions—FLDCW—and for stores if a denormal is being stored). The reorder buffer tag and any exception information is sent back to reorder buffer **32**. At this point, the instructions are complete. However, they are still speculative. When the instructions reach the bottom of reorder buffer **32** (assuming there is no branch misprediction or exception abort), reorder buffer **32** notifies FPU **36** that the instructions should be retired. The speculative state of the floating-point unit **36** is committed to the architectural state when retire queue **316** updates the tags for the architectural register file **328**, and the destination register tags in retire queue **316** are written to the architectural register file **318**.

Convert and classify unit **336** receives all load data, classifies it by data type, and converts it to an internal format if necessary. In one embodiment, convert and classify unit **336** appends a three bit classification tag to each data item. The three bit tag classifies the accompanying data as one of the following eight potential data types: (1) zero, (2) infinity, (3) quiet NaN, (4) signaling NaN, (5) denormal, (6) MMX, (7) normal, or (8) unsupported. NaN is a standard abbreviation for "Not-a-Number". While representations may vary across different implementations, zero data types are typically represented with a zero exponent and a zero significand. Similarly, infinity data types are typically represented with an exponent comprising all asserted ones. A quiet NaN ("QNaN") is generated whenever a floating-point instruction causes an invalid operation, e.g., a square root operation on a negative number. A signaling NaN ("SNaN"), unlike a quiet NaN, generates an exception when used. Signaling NaNs are not generated by FPU **36** and are typically only used by programmers to signal particular error conditions. The table below illustrates the characteristics of each data type for x86 compatible floating-point units (wherein "x" represents either a one or a zero):

Sign	Exponent	Significand	Value
x	$00\ldots00_2$	$0.00\ldots00_2$	Zero
x	$11\ldots11_2$	$1.00\ldots00_2$	Infinity
x	$11\ldots11_2$	$1.1xx\ldots xx_2$	QNaN
x	$11\ldots11_2$	$1.0xx\ldots xx_2$	SNaN
x	$00\ldots00_2$	$0.xx\ldots xx_2$	Denormal

It is noted that these data types may conform to the IEEE-754 specification.

MMX data types are 64 bits wide and comprise either eight packed 8-bit bytes, four packed 16-bit words, or two packed 32-bit double-words. MMX data types may be detected by the MMX opcodes which precede them. Normal data types are standard floating-point values that are either single precision, double precision, or extended precision (before they are translated into an internal data format) and that do not have the characteristics of any of the previously described data types. Unsupported data types are extended precision bit patterns that do not fall into any of the previously described data types and that fall outside of the normal data type as defined by IEEE Standard 754. For example, an extended precision bit sequence having a 0 sign bit, a biased exponent of 11 . . . 11, and a significand in the format (i.ff . . . if) of 0.11 . . . 11 is an unsupported value (wherein "i" is an integer bit and wherein "f" is a fractional bit). Note, however, in other embodiments larger or smaller classification tags and additional or fewer data types may be implemented.

The data types and exemplary formats illustrated above describe the data as it is received and identified by convert and classify unit **336**. Once convert and classify unit **336** classifies the data, the classification tag may be used to identify some or all of the data's properties. For example, if a value is classified as a zero, it may be identified solely by its classification tag instead of having to perform a wide comparison of each bit in the exponent and significand portions of the value. The classification tags may accompany the data throughout FPU **36** and may be stored in register file **18** along with the data.

As discussed above, when data from a load instruction is received by FPU **36**, the data is routed to convert and classify unit **336**. A corresponding reorder buffer tag accompanies the data and is routed to load mapper **338**. As previously noted in the description of microprocessor **10**, the reorder buffer tag identifies the sequence in which out of order instructions should be retired (i.e., committed to architectural state). For load instructions, the reorder buffer tag follows the data from load/store unit **26** to FPU **36**. Load mapper **338** receives the reorder buffer tag and translates it into a physical register tag. The physical register tag indicates which data register within register file **318** the corresponding data is to be loaded into.

Execution and pipeline control unit **340** tracks the status of each stage in execution pipelines **320**, **322**, and **326**. Execution and pipeline control unit **340** contains timing information enabling it to determine the future availability of each execution pipelines. For example, when an FMUL (floating-point multiply) instruction begins execution in multiplication execution pipeline **322**, control unit **340** uses its stored timing information to notify scheduler **314** that the result will be available for output on result bus **346** four clock cycles later. This timing information allows scheduler **314** to efficiently schedule instruction for execution in execution pipelines **320**, **322**, and **326**. Control unit **340** also tracks the status of each pipe stage, receiving and prioritizing exceptions from execution pipelines **320**, **322**, and **326**.

FPU status word, control word, and tag word (collectively, words **344**) are stored within retire queue **316** and indicate which of the status and control registers within register file **318** contain the FPU's current architectural state. For example, in one embodiment register file **318** may comprise 88 registers, i.e., 16 registers to store the current architectural state of FPU **36** (see below), and 72 speculative registers to store the speculative state of FPU **36**. Of the 72 speculative registers, 16 store the "current" speculative state. Of each set of 16 registers that store a particular state of FPU **36** (whether the state is architectural or speculative), eight registers are FPU stack registers and eight registers are micro-architectural registers that store state information that is only accessible to microcode instructions within FPU **36**, i.e., they are not accessible to the programmer and store only internal state information. In one embodiment, each register in register file **314** is 90 bits long, with 87 bits providing storage for internal format data (e.g., one sign bit, 18 exponent bits, and a 68-bit significand) and 3 class bits.

Instruction status register file **342** stores information for execution and pipeline control unit **340**. As a result of

instructions executing out of order in FPU **36**, exceptions that occur within execution pipelines may need to be stored until the instructions generating them are the next to be retired. Retire queue **316** reads instruction status register file **342** when it retires instructions and updates the architectural floating-point status word (FPSW) and floating-point control word (FPCW) and tag word (collectively, **344**) accordingly. This information is conveyed to rename-1 unit along with the current speculative top of stack **352** and on result bus **346**.

Multiplier

FIG. **5** illustrates details of one embodiment of multiplier **50**. In this embodiment, multiplier **50** comprises multiplexers **370** and **372**, adder **374**, booth encoders **376**, booth multiplexers **378**, adder tree **380**, carry-save adders **382** and **384**, carry-propagate adders **386–390**, precision fix-up logic **392** and **394**, storage registers **396–400**, and final result selection multiplexer **402**.

In response to receiving an instruction, multiplexers **370** and **372** are configured to select multiplier and multiplicand operands according to state machine control signals **404A** and **404B**. A state machine (not shown in the figure) controls multiplier **50** and causes multiplexers **370** and **372** to select operands appropriate for the particular instruction being executed. For example, in a standard scalar floating point multiplication, multiplexers **370** and **372** may be configured to select a scalar multiplier operand and a scalar multiplicand operand from memory or other source (e.g. the floating point unit's register stack). However, when performing other types of operations (e.g. square root and division operations) multiplexers **370** and **372** may select operands from other sources such as division/square root ROM **360** or from local bypassing register **396** and **400**. The more complex multiplication operations that entail the use of local bypass register **396** and **400** will be described in greater detail below. The selected multiplier and multiplicand operands are then routed to adder **374** and booth encoders **376** respectively. The output from booth encoders **376** is conveyed to booth multiplexers **378** to select the corresponding partial products.

The selected partial products are then routed to adder tree **380** which comprises a plurality of carry-save adders. These carry-save adders sum the partial products and output a two-component result comprising (i) a carry component and (ii) a sum component. The carry and sum components are conveyed to carry-save adders **382** and **384**. Carry-save adders **382** and **384** are configured to sum the carry and sum components with rounding constants (See FIG. **6**) that assume no overflow and overflow conditions respectively. Thus, carry-save adder **382** sums the carry result, the sum result, and the rounding constant assuming there is no overflow. Similarly, carry-save adder **384** sums the carry result, the sum result, and a rounding constant assuming there is an overflow. Advantageously, by performing the calculation twice in parallel the effect of an overflow condition on processing time is minimized. Note, in some embodiments carry-save adders **382** and **384** form the final stage of adder tree **380**. The results from carry-save adders **382** and **384** are routed in parallel to carry propagate adders **386** and **390**. In parallel, the results from adder tree **380** are passed to carry propagate adder **388**, which includes sticky bit logic to calculate a sticky bit used to round the results from carry propagate adder **386** and **390**. The results from carry propagate adder **388** are provided to final result selection logic **402**, which performs the task of selecting either the result generated assuming an overflow occurred or the result generated assuming an overflow did not occur. The results from carry propagate adders **386** and **390** are conveyed to below precision fix up logic **392** and **394** respectively. Fix up logic **392** and **394** utilize the sticky bit from carry propagate adder **388** to perform corrections assuming an overflow occurred and an overflow did not occur, respectively. These results are provided to final result selection logic **402** via rounding logic **396**. Storage register **400** may be utilized to store the negated version (e.g., the one's complement or two's complement) of the sum of the sum component and carry component from adder tree **380**.

In one embodiment, multiplier **50** may be implemented in static CMOS logic and may have a multiplier latency of four cycles. It may also be configured to operate on a maximum of 76-bit operands (i.e., excluding the exponent bits). Advantageously, this maximum width supports exactly rounded 68-bit division. Multiplier **50** may also be configured to perform all 3DNow!™ SIMD FP multiplication operations. As indicated in the figure, multiplier **50** is preferably implemented in a pipelined fashion to increase overall instruction throughput. One example configuration of pipelines stages is shown in FIG. **5**. Pipeline stage EX1 is the first stage of multiplier **50**. In stage EX1, overlapping groups of four bits of the multiplier operands are inspected are per the booth space three multiplier algorithm. In parallel, adder **374** generates the 3X multiple of the instruction's multiplicand. Booth encoders **376** then generate 26 control signals which control the selection of the 26 booth multiplexers **(378)** to form the appropriately-signed multiples of the multiplicand. This is described in greater detail below.

In the second pipeline stage, EX2, the 26 partial products are reduced to two using binary adder tree **380**. The individual four-two adders of binary tree **380** may also be designed in static CMOS with single rail inputs and outputs. While a static dual rail compressor with dual rail inputs and outputs is typically faster than a single rail configuration, it is also larger and may require twice the routing resources. As discussed further below, adder tree **380** may effectively be implemented as a parallelogram. However, several folding and interleaving techniques may be applied to result in a rectangular adder tree (i.e., thereby leaving internal wire lengths constant).

At the end of the second pipeline stage EX2, the first portion of the multiplier's rounding algorithm is applied. This algorithm involves adding a rounding constant to the sum and parity outputs of adder tree **380**. Since the normalization of the assimilated result may be unknown at this point, the addition is performed twice, i.e., both assuming no overflow occurs and assuming an overflow does occur, in carry-save adders **382** and **384** respectively. Advantageously, carry-save adders **382** and **384** may also be used to implement part of the "back-multiply and subtract" operation (referred to herein as the "BACKMUL" operation) which forms the remainder required for quotient and square root rounding.

The rounding constants used by carry-save adders **382** and **384** are shown in FIG. **6**. Columns **420** through **426** illustrate the rounding constants used for single precision, double precision, extended precision, and internal precision, respectively. Rows **430** through **436** illustrate the rounding constants used for round-to-nearest, round-to-zero, round-to-minus infinity, and round-to-plus infinity rounding modes, respectively. Rows **438** through **442** show the rounding constants for divisions/square root specific operations, LASTMUL, ITERMUL, and BACKMUL operations, respectively. In FIG. **6**, !x implies the bit inversion of x, and 24'b**0** implies 24 zero bits.

In the third pipeline stage EX3, three versions of the carry-assimilated results are formed. Two of these results are rounded results assuming that either overflow or no overflow occurs. The third is the raw unrounded result. In parallel, sticky bit logic within adder **388** examines the low order bits of the sum and carry vectors.

In the fourth pipeline stage, EX4, the bits below the target precision are appropriately cleared. While the rounding constant applied in clock cycle EX2 ensures that the result is correctly rounded, it does not guarantee that the bits below the target least significant bit (LSB) are cleared, as may be required for x87 compatibility. The LSB is then conditionally inverted for regular multiplication operations (using the round-to-nearest rounding mode) as a function of the sticky bit to correctly handle the nearest-even case. Finally, the most significant bit (MSB) of the unrounded result determines whether or not overflow has occurred, and it selects between the no overflow and the with overflow rounded results. For division and square root iterations, an extra result R_i is also provided. R_i is the one's complement of the regular unrounded multiply result for division (and may also be used as an approximation to 3−N+2 for the square root calculations). Both of the results are available for local storage and bypassing for use within division and square root iterations. In the case of the last cycle of a division or square root instruction, the BACKMUL instruction is performed by appropriately rounding the results chosen from the previously computed target results; the quotient ("Q"), the quotient plus one ("Q+1"), or the quotient minus one ("Q−1").

The unrounded result may also be used in the case of IEEE "tiny" results with the underflow exception masked. A tiny number occurs when the computed rounded result is below the minimum extended precision normal number. In this case, the unrounded result may be passed to a microcode handling routine which may properly denormalize and round the result for x87 compatibility. The unrounded result may also be used to determine whether round up has occurred for proper setting of the "C1" condition code bit in the floating point status word (i.e., bit **8**). Roundup occurs when the rounded result differs from the unrounded result, and the C1 bit can be set appropriately given this information. For division and square root operations, the unrounded result is synthesized by appropriately choosing between Q−1 and Q.

Early Completion

In some embodiments, multiplier **50** may advantageously be configured to complete particular types of operations in fewer clock cycles than standard operations. One example of this is division by a number that is exactly a power of two. In theory, division by exact powers of two amounts to a simple reduction in the exponent leaving the significand unchanged. Accordingly, multiplier **50** may be configured to support variable-latency instructions. For divide-by power-of-two operations, the exponent adjustment occurs during the initial approximation lookup phase, while the rounding of the significand occurs using the LASTMUL and BACKMUL operations. Microprocessor **10** and multiplier **50** may be configured such that the latency of these operations are acceptable with respect to the notification time to the floating point unit's scheduler **314**. The same approach may be utilized for multiplication by powers of two.

FIGS. 7–8

FIG. **7** illustrates a block diagram illustrating one embodiment of multiplier **50** utilizing a parallelogram configuration for adder tree **380**. As shown in the figure, the adder tree may be perform multiply a 76-bit multiplier operand and a 76-bit multiplicand operand, or a 24–32 bit packed floating point multiplier and a 24–32 bit packed floating point multiplicand.

FIG. **8** illustrates more details of one embodiment of the multiplier from FIG. **7**. In this embodiment, selection logic **450** is configured to allow multiplier **50** to select and multiply (a) two packed floating point operands, wherein each packed floating point operand comprises two or more individual floating point values; or (b) two scalar floating point operands, wherein each scalar operand comprises a single floating point value. As noted above, selection logic **450** also can be configured to select the results from adder tree **380** as an input for iterative multiplication and square root operations. As shown in the figure, selection logic **450** comprises a plurality of multiplexers configured to select from among the original source operands A and B, the results from a previous iterative operation, and constant ROM **358**. As previously noted, state machine **452** comprises the control logic to direct multiplier **50** to perform standard multiplication operations, packed multiplication operations, and iterative calculations such as divide and square root.

Rounding

FIG. **9** illustrates more details of one embodiment of multiplier **50** and the pipeline stages EX2–EX4. As shown in the figure, the results from adder tree **380** are conveyed to carry-save adder ("CSA") **382**, CSA **384**, and carry-propagate adder ("CPA") **388**. CSA **382** receives three input operands, the sum operand from adder tree **380**, the carry component of the result from adder tree **380**, and third input selected by multiplexer **452**A. Multiplexer **452**A is configured to select either rounding constant **450**A (as described in the table of FIG. **6**), or the original source A operand from the floating point instruction. Multiplexer **452**B is configured to select either rounding constant **450**B shifted one bit (as described in the table of FIG. **6**), or the original source A operand from the floating point instruction, also shifted one bit.

Special Instructions

As previously noted, multiplier **50** may also be configured to execute a number of special instructions. For example, some of these instructions are components of iterative calculations such as divide and square root. Embodiments of these special instructions will be described below.

Itermul (A,B)

This is a multiplication operation of A and B that forces the rounding to be round to nearest. It assumes that each of the input operands are 76 bits wide and that the intermediate results (which is 152 bits wide) is rounded to 76 bits precision. This wider precision may be useful to correct the uncorrected rounding errors which accumulate throughout the iterations.

Lastmul (A, B, PC)

This is a multiplication operation of A and B that forces the rounding to be round to nearest. It performs the rounding to a precision 1 bit wider than the target precision specified by PC. For division, just prior to the rounding of this operation, the double with product "Q" prime may be required to be accurate to at least:

$-2^{-(pc+2)}21\ Q-Q'<2^{-(pc+2)}$

where Q is the infinitely precise quotient, and pc is the target precision in bits, wherein 1 ulp (unit in the last place) for such a pc bit number is $2^{-(pc-1)}$. Exact rounding may require that the result have an error no worse than ±0.5 ulp, or 2^{31} PC. Multiplier **50** may be configured to utilize a rounding algorithm that ensures that the result is computed to an accuracy of at least one additional bit. This is because the final quotient result may be in the range of (0.5,2). Thus, 1 bit of normalization may be used, resulting in the potential need for the one additional bit of accuracy. For square root operations, just prior to the rounding the double-width product S' may be required to be accurate to at least:

$$-2^{-(pc+1)}<S-S'<2^{-(pc+1)},$$

wherein S is the infinitely precise square root, and pc is the target precision in bits. Since the square root result is in the range [1,2), it may have a looser constraint on the accuracy of the input to this operation.

After rounding and normalizing to pc +1 bits through the LASTMUL op, the resulting value R" may satisfy:

$$-2^{-pc}<R-R"<2^{-pc},$$

wherein R is either the infinitely precise quotient or square root as appropriate. Thus, the value may have an error of (−0.5,+0.5) ulp with respect to the final pc bit number.

Backmul(B,Q,A)

This is a multiplication operation that operates on two source operands B and Q, and it also accepts a third operand A. The 152-bit intermediate product of the two sources in carry-save form is added with an inverted version of the third operand, with the low-order bits filled with 1's as shown in the table in FIG. **6**. These three values are then input into rounding carry-save adders **382** and **384**, with the unused lsb carry bit set to one, realizing the function of BxQ+TwosComp(A). This implements the negative version of the back multiply and subtraction operation to form the remainder, that is:

$$B\times Q-A$$

is implemented instead of the desired

$$A-B\times Q.$$

The sign of this remainder is thus the negative of the true remainder sign. This operation returns two bits of status: whether the sign of the remainder is negative, taken from a high order bit of the result, and whether the remainder is exactly zero, using fast sticky-bit logic. Since Q could be rounded to any of four precisions, the high order bit is chosen high enough to allow it to suffice for all of the precisions.

Comp1(x)

This operation returns the one's complement by bit inversion of the unrounded version of the currently-computed product. The unbiased exponent is forced to either −1 or 0 depending upon whether the result should be in the binade [0.5,1) or [1,2). Using the one's complement instead of the two's complement in the iterations may add a small amount of error in some embodiments. However, this error may be taken into account when designing the width of the multiplier and the required accuracy of any initial approximations (e.g., from constant ROM **360**).

Comp3(x)

This operation returns an approximation of (3−x)/2 for the unrounded version of the currently-computed product. This approximation may be formed by bit inversion and shifting.

Round(qi,rem,pc,rc)

This rounding function may be configured to assume that a biased trial result qi has been computed with pc +1 bits, which is known to have an error of (−0.5,+0.5) ulp with respect to pc bits. The extra bit, or guard bit, may be used along with the sign of the remainder, a bit stating whether the remainder is exactly zero, and rc to choose which of the three possible results, q, q−1, or q+1 are equal to qi truncated to pc bits and decremented or incremented appropriately. One embodiment of the actions taken for the round function are shown in FIG. **10**. The rounding details are shown in FIG. **10**. In the figure, RN represents round-to-nearest, RNE represents round-to-nearest-even, RP represents round to positive infinity, RM represents round to minus infinity, and RZ represents round-to-zero.

For RN, in the case of an exactly halfway situation, it may be necessary to inspect L, the least significant bit of q to determine the action. If L=0, then the result is correctly rounded to nearest-even. Otherwise, the result is incremented to the closest even result. It should be noted that for division where the precision of the input operands is the same as that of the result, the exact halfway case will not occur. Only when the result precision is smaller than the inputs can such a result occur and rounding be required. For the directed rounding modes RP and RM, the action may depend upon the sign of the quotient estimate. Those entries that contain two operations such as pos/neg are for the sign of the final result itself being positive and negative respectively.

This function, along with the computation of q−1 and q+1, may be implemented in multiplier **50** in parallel with the BACKMUL operation. The status information from the BACKMUL op may be used as input to the ROUND function to quickly choose and return the correctly-rounded result.

Additional details regarding a number of the features of different embodiments of multiplier **50** (e.g., the use of the special instructions, division, square root operations, and rounding) may be found in the parent applications cited at the beginning of this application.

Computer System

Turning now to FIG. **11**, a block diagram of one embodiment of a computer system **500** configured to utilize one embodiment microprocessor **10** with multiplier **50** is shown. Microprocessor **10** is coupled to a variety of system components through a bus bridge **502**. Other embodiments are possible and contemplated. In the depicted system, a main memory **504** is coupled to bus bridge **502** through a memory bus **506**, and a graphics controller **508** is coupled to bus bridge **502** through an AGP bus **510**. Finally, a plurality of PCI devices **512A**–**512B** are coupled to bus bridge **502** through a PCI bus **514**. A secondary bus bridge **516** may further be provided to accommodate an electrical interface to one or more EISA or ISA devices **518** through an EISA/ISA bus **520**. Microprocessor **10** is coupled to bus bridge **502** through a CPU bus **524**.

Bus bridge **502** provides an interface between microprocessor **10**, main memory **504**, graphics controller **508**, and devices attached to PCI bus **514**. When an operation is

received from one of the devices connected to bus bridge **502**, bus bridge **502** identifies the target of the operation (e.g. a particular device or, in the case of PCI bus **514**, that the target is on PCI bus **514**). Bus bridge **502** routes the operation to the targeted device. Bus bridge **502** generally translates an operation from the protocol used by the source device or bus to the protocol used by the target device or bus.

In addition to providing an interface to an ISA/EISA bus for PCI bus **514**, secondary bus bridge **516** may further incorporate additional functionality, as desired. For example, in one embodiment, secondary bus bridge **516** includes a master PCI arbiter (not shown) for arbitrating ownership of PCI bus **514**. An input/output controller (not shown), either external from or integrated with secondary bus bridge **516**, may also be included within computer system **500** to provide operational support for a keyboard and mouse **522** and for various serial and parallel ports, as desired. An external cache unit (not shown) may further be coupled to CPU bus **524** between microprocessor **10** and bus bridge **502** in other embodiments. Alternatively, the external cache may be coupled to bus bridge **502** and cache control logic for the external cache may be integrated into bus bridge **502**.

Main memory **504** is a memory in which application programs are stored and from which microprocessor **10** primarily executes. A suitable main memory **504** comprises DRAM (Dynamic Random Access Memory), and preferably a plurality of banks of SDRAM (Synchronous DRAM).

PCI devices **512A–512B** are illustrative of a variety of peripheral devices such as, for example, network interface cards, video accelerators, audio cards, hard or floppy disk drives or drive controllers, SCSI (Small Computer Systems Interface) adapters and telephony cards. Similarly, ISA device **518** is illustrative of various types of peripheral devices, such as a modem, a sound card, and a variety of data acquisition cards such as GPIB or field bus interface cards.

Graphics controller **508** is provided to control the rendering of text and images on a display **526**. Graphics controller **508** may embody a typical graphics accelerator generally known in the art to render three-dimensional data structures which can be effectively shifted into and from main memory **504**. Graphics controller **508** may therefore be a master of AGP bus **510** in that it can request and receive access to a target interface within bus bridge **502** to thereby obtain access to main memory **504**. A dedicated graphics bus accommodates rapid retrieval of data from main memory **504**. For certain operations, graphics controller **508** may further be configured to generate PCI protocol transactions on AGP bus **510**. The AGP interface of bus bridge **502** may thus include functionality to support both AGP protocol transactions as well as PCI protocol target and initiator transactions. Display **526** is any electronic display upon which an image or text can be presented. A suitable display **526** includes a cathode ray tube ("CRT"), a liquid crystal display ("LCD"), etc.

It is noted that, while the AGP, PCI, and ISA or EISA buses have been used as examples in the above description, any bus architectures may be substituted as desired. It is further noted that computer system **500** may be a multiprocessing computer system including additional microprocessors (e.g. microprocessor **10***a* shown as an optional component of computer system **500**). Microprocessor **10***a* may be similar to microprocessor **10**. More particularly, microprocessor **10***a* may be an identical copy of microprocessor **10**. Microprocessor **10***a* may share CPU bus **524** with microprocessor **10** or may be connected to bus bridge **502** via an independent bus.

It is still further noted that the present discussion may refer to the assertion of various signals. As used herein, a signal is "asserted" if it conveys a value indicative of a particular condition. Conversely, a signal is "deasserted" if it conveys a value indicative of a lack of a particular condition. A signal may be defined to be asserted when it conveys a logical zero value or, conversely, when it conveys a logical one value. Additionally, various values have been described as being discarded in the above discussion. A value may be discarded in a number of manners, but generally involves modifying the value such that it is ignored by logic circuitry which receives the value. For example, if the value comprises a bit, the logic state of the value may be inverted to discard the value. If the value is an n-bit value, one of the n-bit encodings may indicate that the value is invalid. Setting the value to the invalid encoding causes the value to be discarded. Additionally, an n-bit value may include a valid bit indicative, when set, that the n-bit value is valid. Resetting the valid bit may comprise discarding the value. Other methods of discarding a value may be used as well.

Although the embodiments above have been described in considerable detail, other versions are possible. Numerous variations and modifications will become apparent to those skilled in the art once the above disclosure is fully appreciated. It is intended that the following claims be interpreted to embrace all such variations and modifications.

What is claimed is:

1. A method for performing integer and floating point multiplication, the method comprising:
   - detecting a type of multiplication operation to be performed;
   - asserting one or more control signals in response to detecting the type of multiplication operation to be performed;
   - conveying the asserted control signals to a multiplier and selection logic configurable to perform the multiplication operation;
   - selecting a multiplier operand and a multiplicand operand corresponding to the multiplication operation to be performed in response to the asserted control signals;
   - multiplying the multiplier operand and the multiplicand operand, wherein the multiplication operation is selected from the group comprising: scalar floating point multiplication operations, iterative multiplication operations performed in the context of a divide operation, and iterative multiplication operations performed in the context of a square root operation;
   - detecting floating point divide operations that have divisors that are exactly a power of two in response to detecting an iterative multiplication operation associated with a divide operation; and
   - executing the divide by power of two operation by:
     - shifting dividends; and
     - rounding the result to the appropriate precision using a specified rounding mode.

2. A method for performing integer and floating point multiplication, the method comprising:
   - detecting a type of multiplication operation to be performed;
   - asserting one or more control signals in response to detecting the type of multiplication operation to be performed;
   - conveying the asserted control signals to a multiplier and selection logic configurable to perform the multiplication operation;

selecting a multiplier operand and a multiplicand operand corresponding to the multiplication operation to be performed in response to the asserted control signals;

multiplying the multiplier operand and the multiplicand operand, wherein the multiplication operation is selected from the group comprising: scalar floating point multiplication operations, iterative multiplication operations performed in the context of a divide operation, and iterative multiplication operations performed in the context of a square root operation;

wherein multiplying includes:

generating a plurality of partial products;

summing the partial products using an adder tree to generate a sum component and a carry component;

summing the sum component and the carry component assuming there is no overflow to generate a first result;

summing the sum component and the carry component assuming there is an overflow to generate a second result; and

selecting either the first result or the second result based on a most significant bit of either the first result or the second result.

3. The method as recited in claim **2**, wherein said multiplying further comprises:

selectively adding a rounding constant to the first result and the second result.

4. A floating point unit comprising:

a multiplier configured to perform scalar floating point multiplication of the form A×B,

wherein the multiplier includes:

a binary adder tree configured to generate a result having a sum component and a carry component; and

an overflow adder coupled to the binary adder tree and configured to sum the sum component and the carry component from the adder tree with a first rounding constant assuming the resulting sum will be normalized and have an overflow;

a non-overflow adder coupled to the binary adder tree and configured to sum the sum component and the carry component from the adder tree with a second rounding constant assuming the resulting sum will be normalized and not have an overflow; and

a non-rounding adder coupled to the binary adder tree and configured to sum the sum component and the carry component from the adder tree assuming the result is not be rounded.

5. The floating point unit as recited in claim **4**, wherein the binary adder tree comprises a plurality of cascaded carry-save adders, and wherein each of the overflow adder, the non-overflow adder, and the non-rounding adder is a carry-propagate adder.

6. The floating point unit as recited in claim **4**, wherein the multiplier is further configured to perform scalar floating point operations of the form X×Y−Z. wherein X, Y and Z are floating point operands, and wherein the product X×Y is either (i) in the same binate as Z, or (ii) one binate greater than Z, wherein the multiplier further comprises one or more inverters configured to generate the one's complement of Z, wherein the overflow adder and the non-overflow adder are configured to receive and sum (a) the one's complement of Z, (b) the product X×Y, and (c) a constant one.

7. The floating point unit as recited in claim **6**, wherein X and Y are equal for square root instructions.

8. The floating point unit as recited in claim **6**, wherein X and Y need not be equal for divide instructions.

9. The floating point unit as recited in claim **4**, further comprising selection logic configured to select the results from the non-rounding adder if the results from either the overflow adder or the non-overflow adder is determined to be below a predetermined minimum value.

10. The floating point unit as recited in claim **9**, wherein the multiplier is configured to call a microcode routine upon selecting the results from the non-rounding adder in response to a masked bit being set, wherein the microcode routine is configured to denormalize the results from the non-rounding adder, round the denormalized result, and then normalize the rounded denormalized result.

11. The floating point unit as recited in claim **4** further comprising selection logic configured to select two source operands for the multiplier to multiply.

12. The floating point unit as recited in claim **11**, wherein the source operands are selected from at least the following:

(a) original source operands including: two packed floating point values, wherein each packed floating point value comprises two or more individual floating point values, and two scalar floating point values; and

(b) results stored in a result register.

13. The floating point unit as recited in claim **12**, wherein the multiplier further comprises a state machine coupled to the selection logic, wherein the state machine is configured to cause the selection logic to select the stored results in response to the multiplier executing a division instruction by performing iterative multiplication operations.

14. The floating point unit as recited in claim **12**, wherein the multiplier further comprises a state machine coupled to the selection logic, wherein the state machine is configured to cause the selection logic to select the stored results in response to the multiplier executing a square root instruction by performing iterative multiplication operations.

15. The floating point unit as recited in claim **12**, wherein the selection logic comprises a first multiplexer unit configured to select a multiplier operand for the multiplier, and a second multiplexer unit configured to select a multiplicand operand for the multiplier, wherein the first multiplexer unit is configured to select from at least the original source operands and the stored results.

16. A computer system comprising:

a processor;

a main system memory coupled to the processor via a bus;

wherein the processor includes a floating point unit including:

a multiplier configured to perform floating point multiplication of the form A×B;

a register configured to store results generated by the multiplier; and

selection logic configured to select source operands for the multiplier to multiply,

wherein the multiplier includes:

a binary adder tree configured to generate a result having a sum component and a carry component; and

an overflow adder coupled to the binary adder tree and configured to sum the sum component and the carry component from the adder tree with a first rounding constant assuming the resulting sum will be normalized and have an overflow;

a non-overflow adder coupled to the binary adder tree and configured to sum the sum component and the carry component from the adder tree with

a second rounding constant assuming the resulting sum will be normalized and not have an overflow; and

a non-rounding adder coupled to the binary adder tree and configured to sum the sum component and the carry component from the adder tree assuming the result is not be rounded.

17. The computer system as recited in claim **16**, wherein the source operands are selected from at least the following:

(a) original source operands including: two packed floating point values, wherein each packed floating point value comprises two or more individual floating point values, and two scalar floating point values; and

(b) the results stored in the register.

18. The computer system as recited in claim **17**, wherein the multiplier further comprises a state machine coupled to the selection logic, wherein the state machine is configured to cause the selection logic to select the stored results in response to the multiplier executing a division instruction by performing iterative multiplication operations.

19. The computer system as recited in claim **17**, wherein the multiplier further comprises a state machine coupled to the selection logic, wherein the state machine is configured to cause the selection logic to select the stored results in response to the multiplier executing a square root instruction by performing iterative multiplication operations.

20. The computer system as recited in claim **16**, wherein the selection logic comprises a first multiplexer unit configured to select a multiplier operand for the multiplier, and a second multiplexer unit configured to select a multiplicand operand for the multiplier, wherein the first multiplexer unit is configured to select from at least the original source operands and the stored results.

* * * * *

US006535946B1

(12) **United States Patent**
Bryant et al.

(10) **Patent No.: US 6,535,946 B1**
(45) **Date of Patent: Mar. 18, 2003**

(54) **LOW-LATENCY CIRCUIT FOR SYNCHRONIZING DATA TRANSFERS BETWEEN CLOCK DOMAINS DERIVED FROM A COMMON CLOCK**

(75) Inventors: **Christopher D. Bryant**, Austin, TX (US); **Robin W. Edenfield**, Dallas, TX (US)

(73) Assignee: **National Semiconductor Corporation**, Santa Clara, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/477,321**

(22) Filed: **Jan. 4, 2000**

(51) **Int. Cl.**[7] **G06F 13/00**; H04L 7/00

(52) **U.S. Cl.** **710/305**; 375/354; 713/400; 327/141

(58) **Field of Search** 710/305; 713/400; 375/354; 327/141, 144, 145, 146; 325/147

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,022,057 A	*	6/1991	Nishi et al.	375/373
5,132,990 A	*	7/1992	Dukes	375/362
5,487,092 A	*	1/1996	Finney et al.	375/354
5,680,594 A	*	10/1997	Charneski et al.	713/400
6,175,603 B1	*	1/2001	Chapman et al.	375/354
6,239,626 B1	*	5/2001	Chesavage	327/147

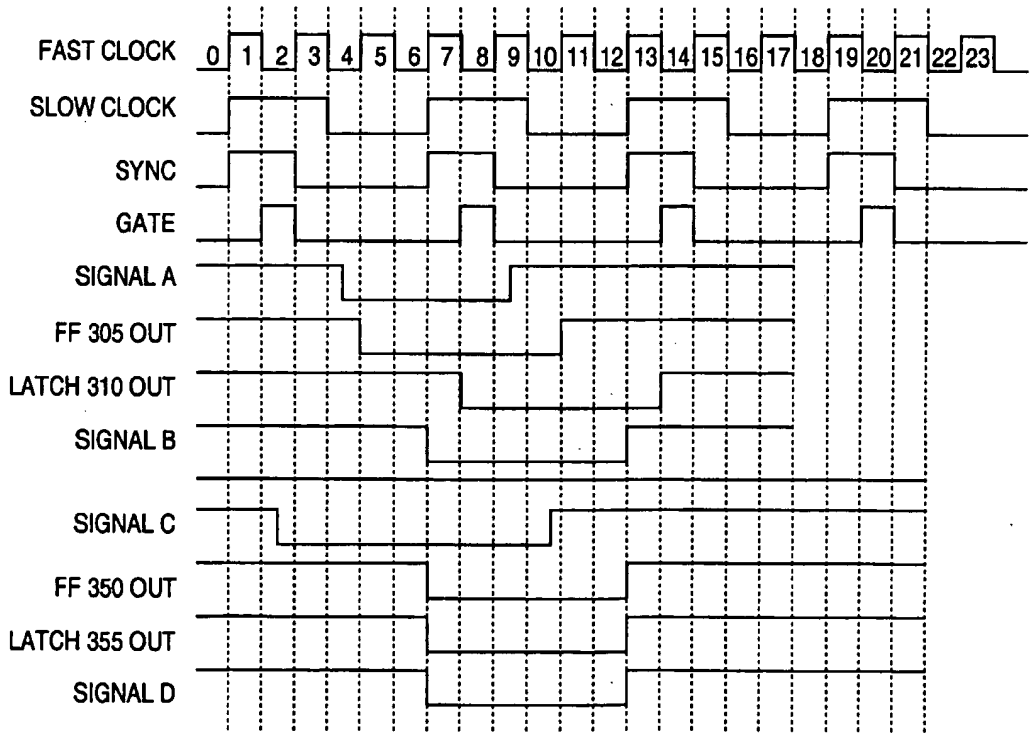
* cited by examiner

Primary Examiner—Xuan Thai
(74) *Attorney, Agent, or Firm*—Davis Munck, P.C.

(57) **ABSTRACT**

There is disclosed, for use in an x86-compatible processor, an interface circuit for synchronizing the transfer of signals between different clock domains derived from a common core clock, where the phase and frequency relationships between the different domain clocks are known. The interface circuit comprises 1) a first latch having a data input for receiving a data signal from the first clock domain, a clock input for receiving the first clock signal, and an output; 2) a second latch having a data input coupled to the first latch output, an enable input for receiving a gating signal, a clock input for receiving the first clock signal, and an output; 3) a third latch having a data input for receiving the data signal, an enable input for receiving a gating signal, a clock input for receiving the first clock signal, and an output; and 4) a multiplexer having a first data input coupled to the second latch output, a second data input coupled to the third latch output, and a selector input for selecting one of the first data input and the second data input for transfer to an output of the multiplexer.

20 Claims, 6 Drawing Sheets



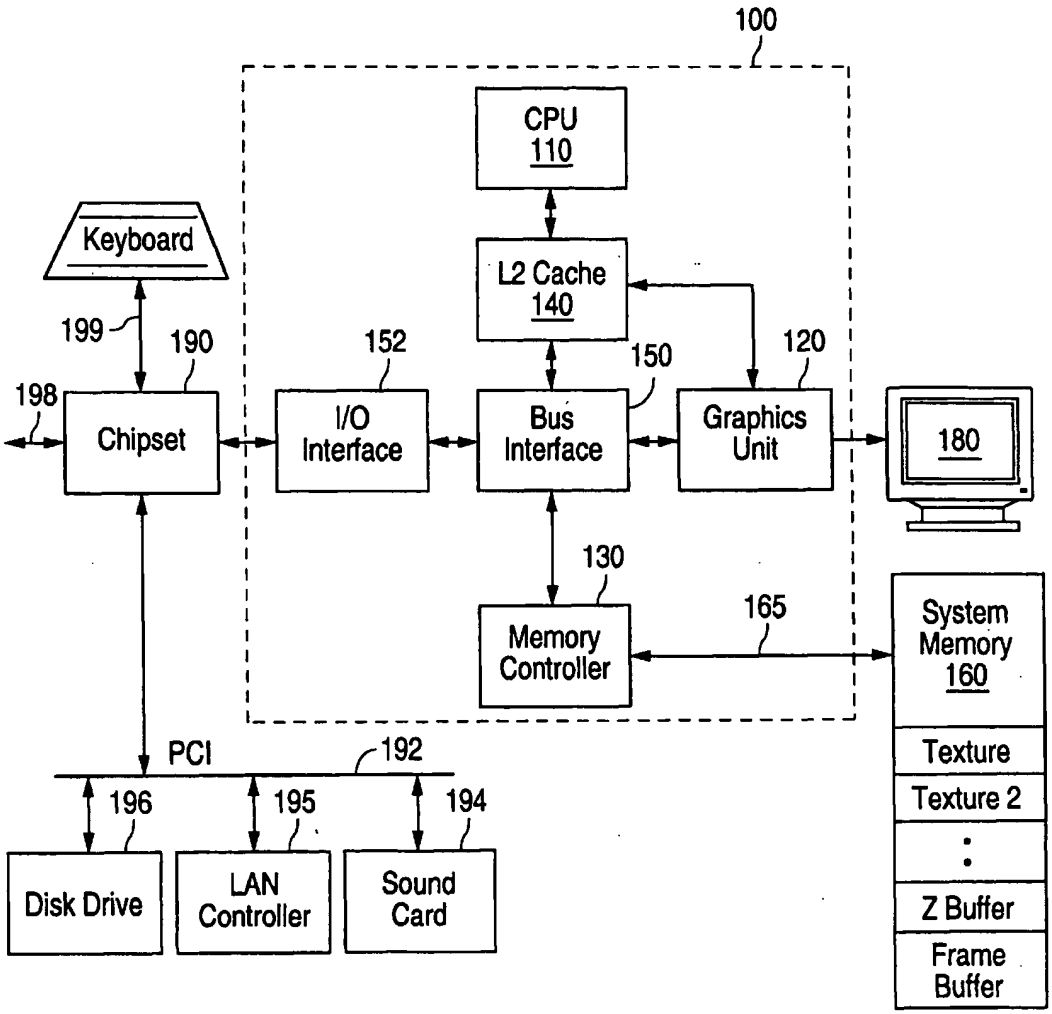

100
CPU
110
L2 Cache
140
Keyboard
199
190
152
150
120
198
Chipset
I/O
Interface
Bus
Interface
Graphics
Unit
180
130
165
Memory
Controller
System
Memory
160
Texture
Texture 2
Z Buffer
Frame
Buffer
PCI
192
196
195
194
Disk Drive
LAN
Controller
Sound
Card

FIG. 1

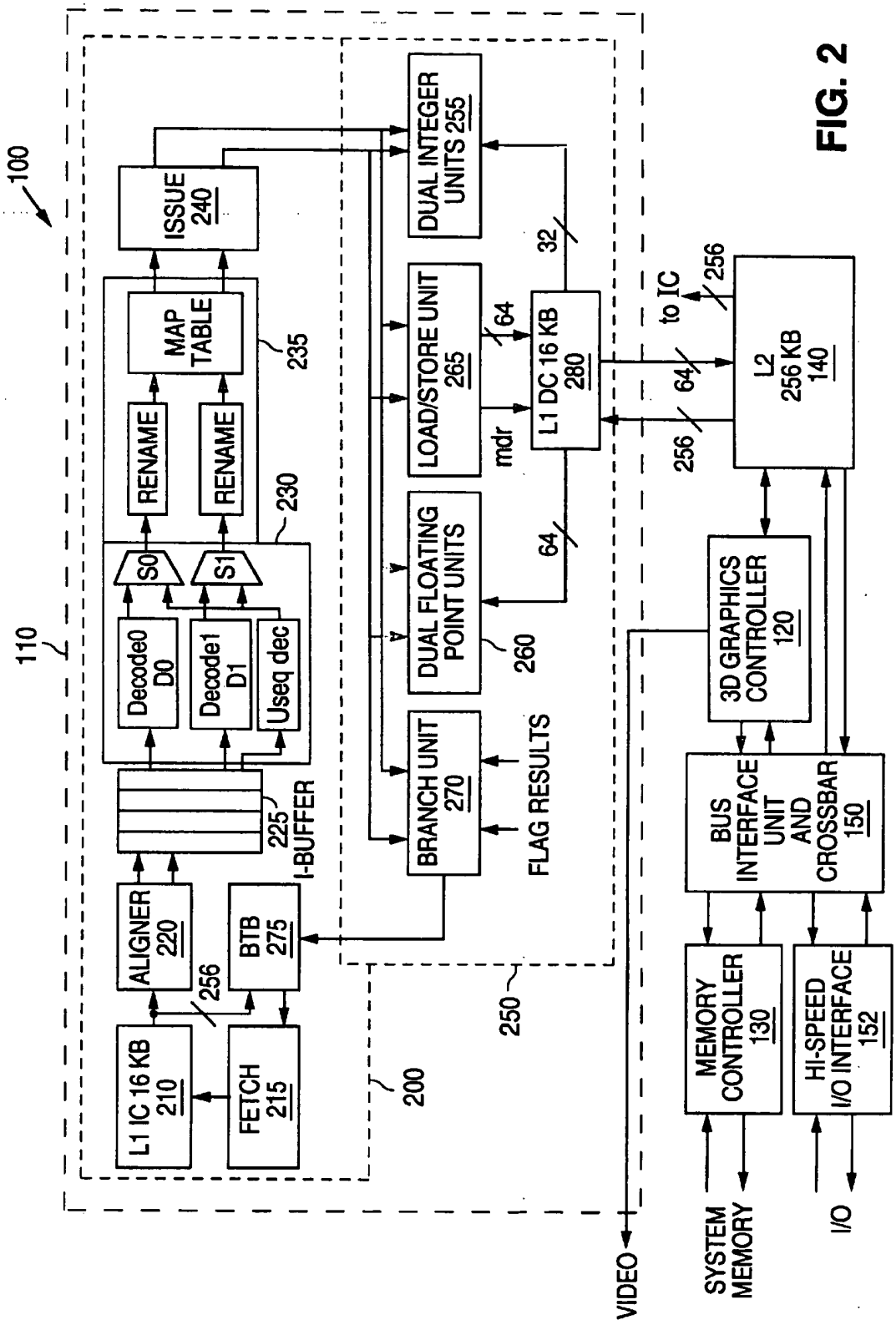

100
110
200
250
ISSUE 240
MAP TABLE
235
RENAME
RENAME
230
S0
S1
Decode0 D0
Decode1 D1
Useq dec
225 I-BUFFER
ALIGNER 220
256
BTB 275
L1 IC 16 KB 210
FETCH 215
DUAL INTEGER UNITS 255
32
LOAD/STORE UNIT 265
64
mdr
L1 DC 16 KB 280
DUAL FLOATING POINT UNITS
260
64
BRANCH UNIT 270
FLAG RESULTS
VIDEO
to IC
256
L2 256 KB 140
64
256
3D GRAPHICS CONTROLLER 120
BUS INTERFACE UNIT AND CROSSBAR 150
MEMORY CONTROLLER 130
HI-SPEED I/O INTERFACE 152
SYSTEM MEMORY
I/O

FIG. 2

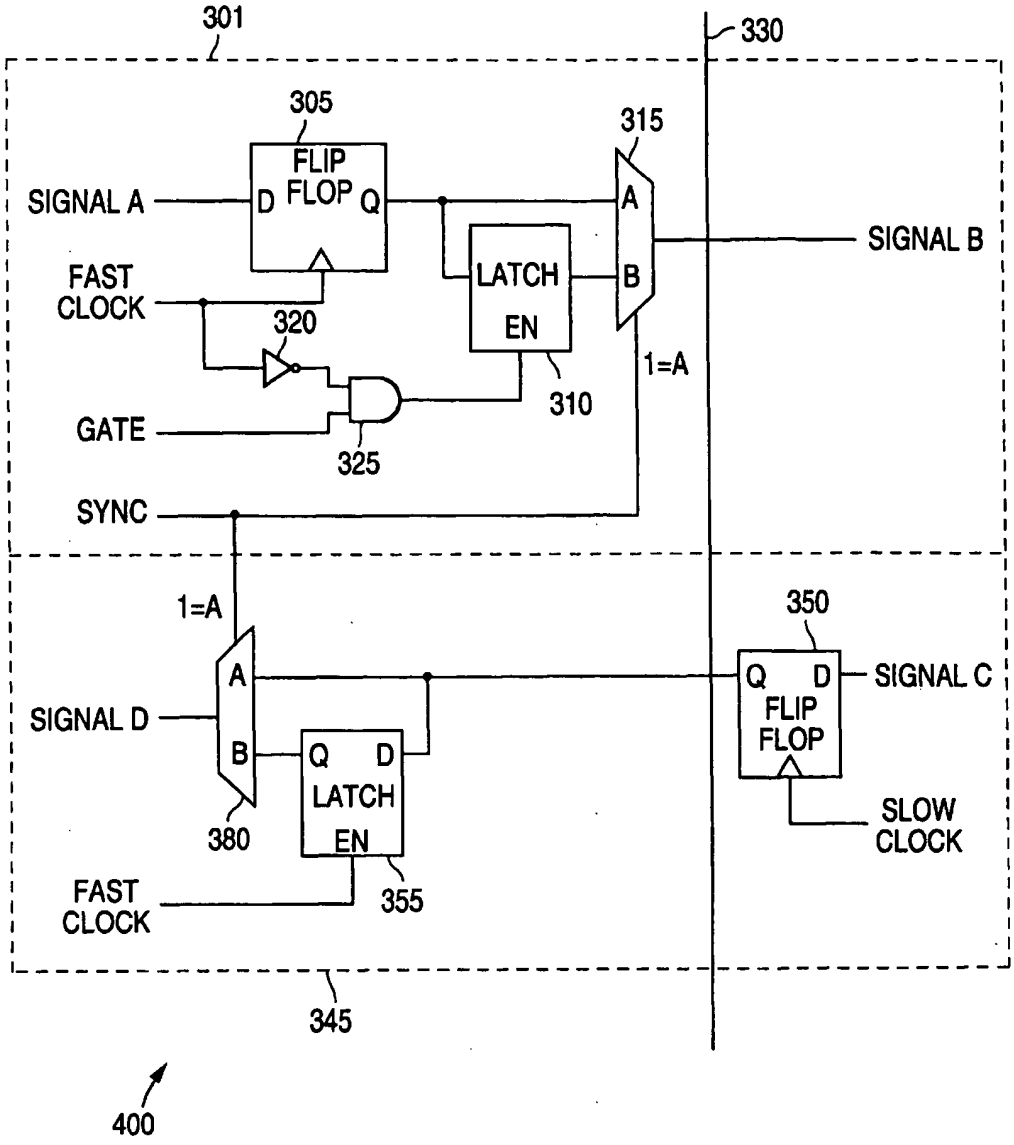

301
330
305
315
FLIP
FLOP
SIGNAL A
D
Q
A
SIGNAL B
LATCH
B
FAST
CLOCK
EN
320
1=A
310
GATE
325
SYNC
1=A
350
A
Q
D
SIGNAL C
SIGNAL D
B
FLIP
FLOP
Q
D
LATCH
SLOW
CLOCK
380
EN
FAST
CLOCK
355
345
400

FIG. 3

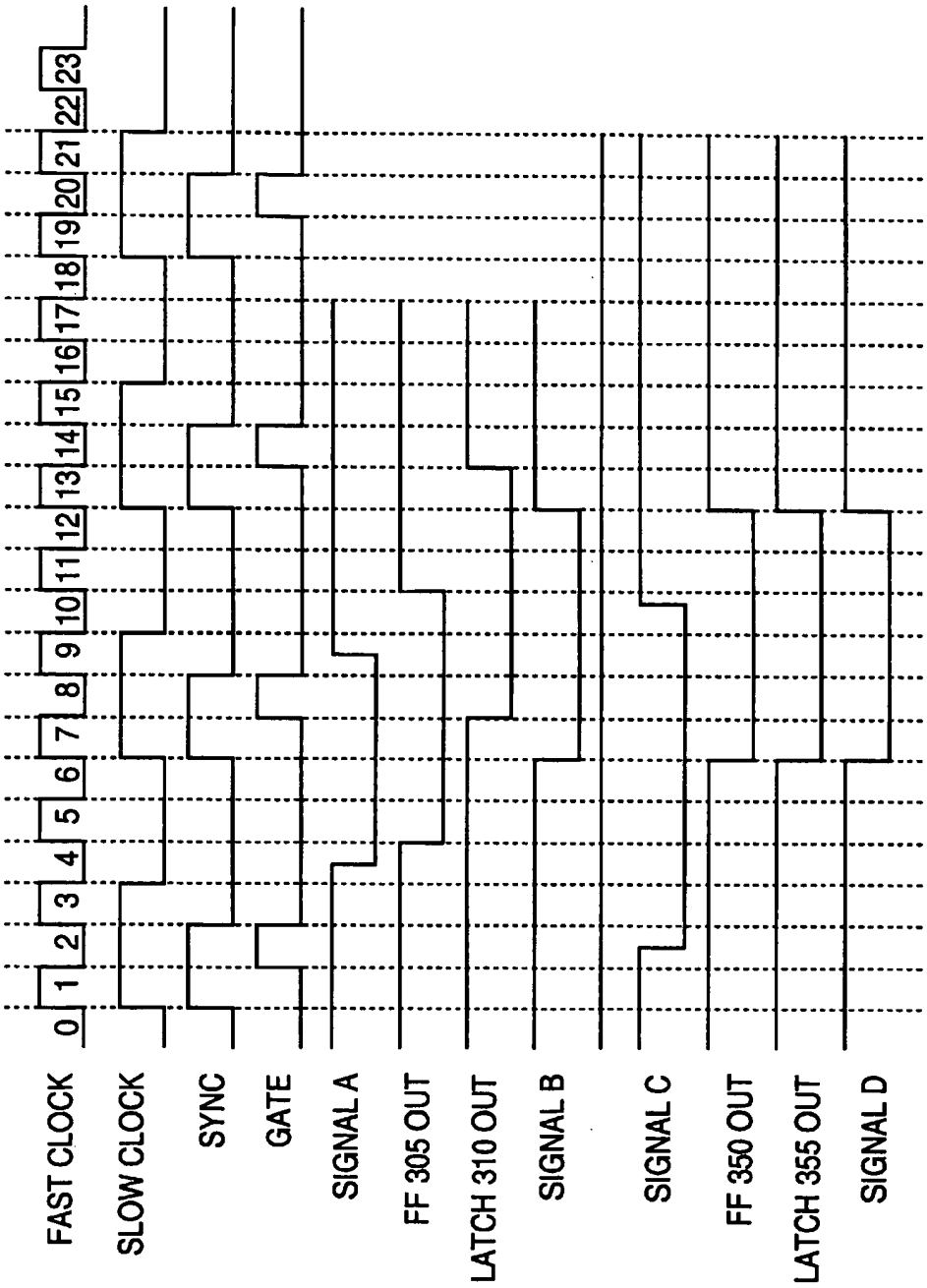

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
FAST CLOCK
SLOW CLOCK
SYNC
GATE
SIGNAL A
FF 305 OUT
LATCH 310 OUT
SIGNAL B
SIGNAL C
FF 350 OUT
LATCH 355 OUT
SIGNAL D

FIG. 4

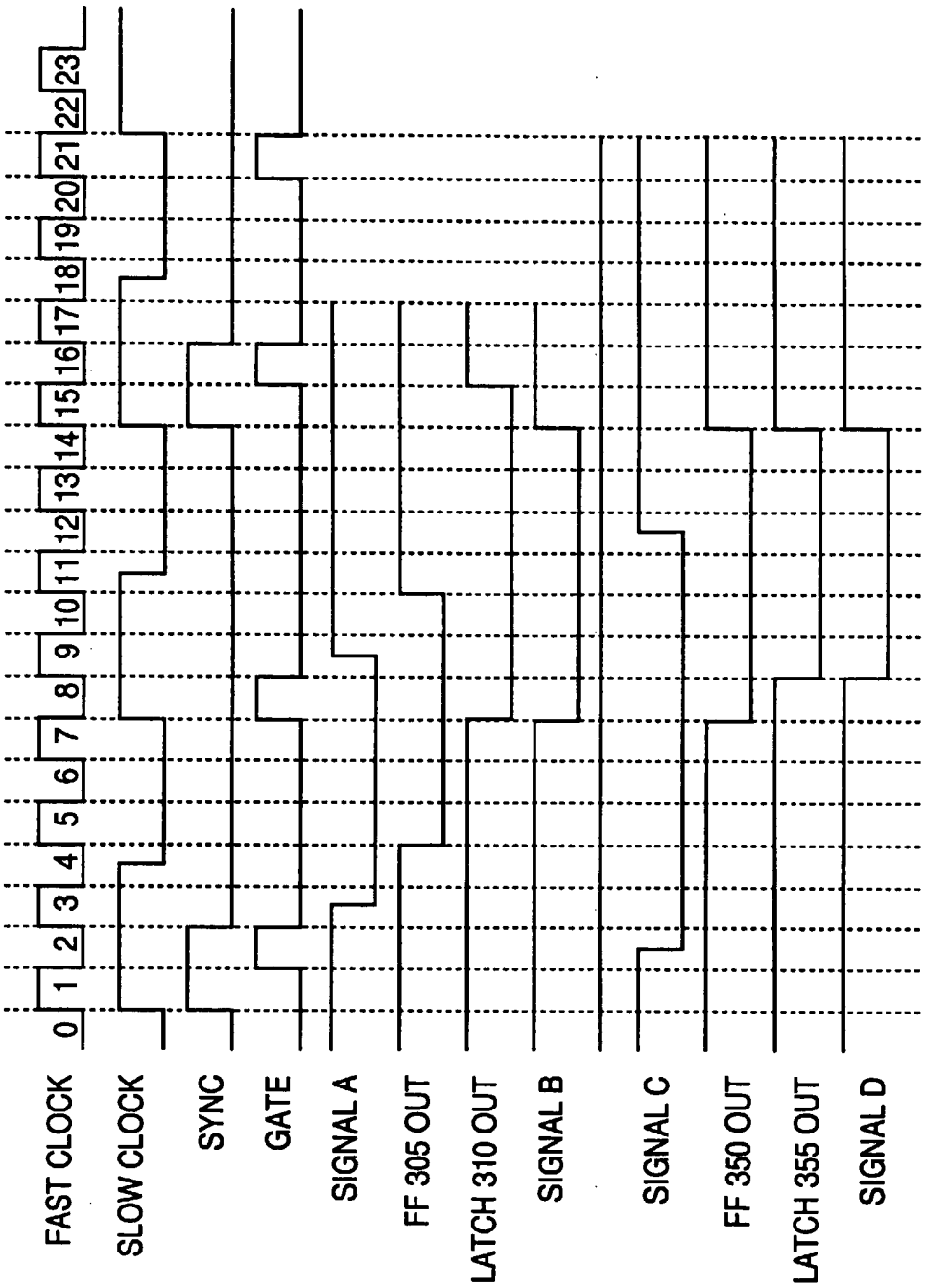

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
FAST CLOCK
SLOW CLOCK
SYNC
GATE
SIGNAL A
FF 305 OUT
LATCH 310 OUT
SIGNAL B
SIGNAL C
FF 350 OUT
LATCH 355 OUT
SIGNAL D

FIG. 5

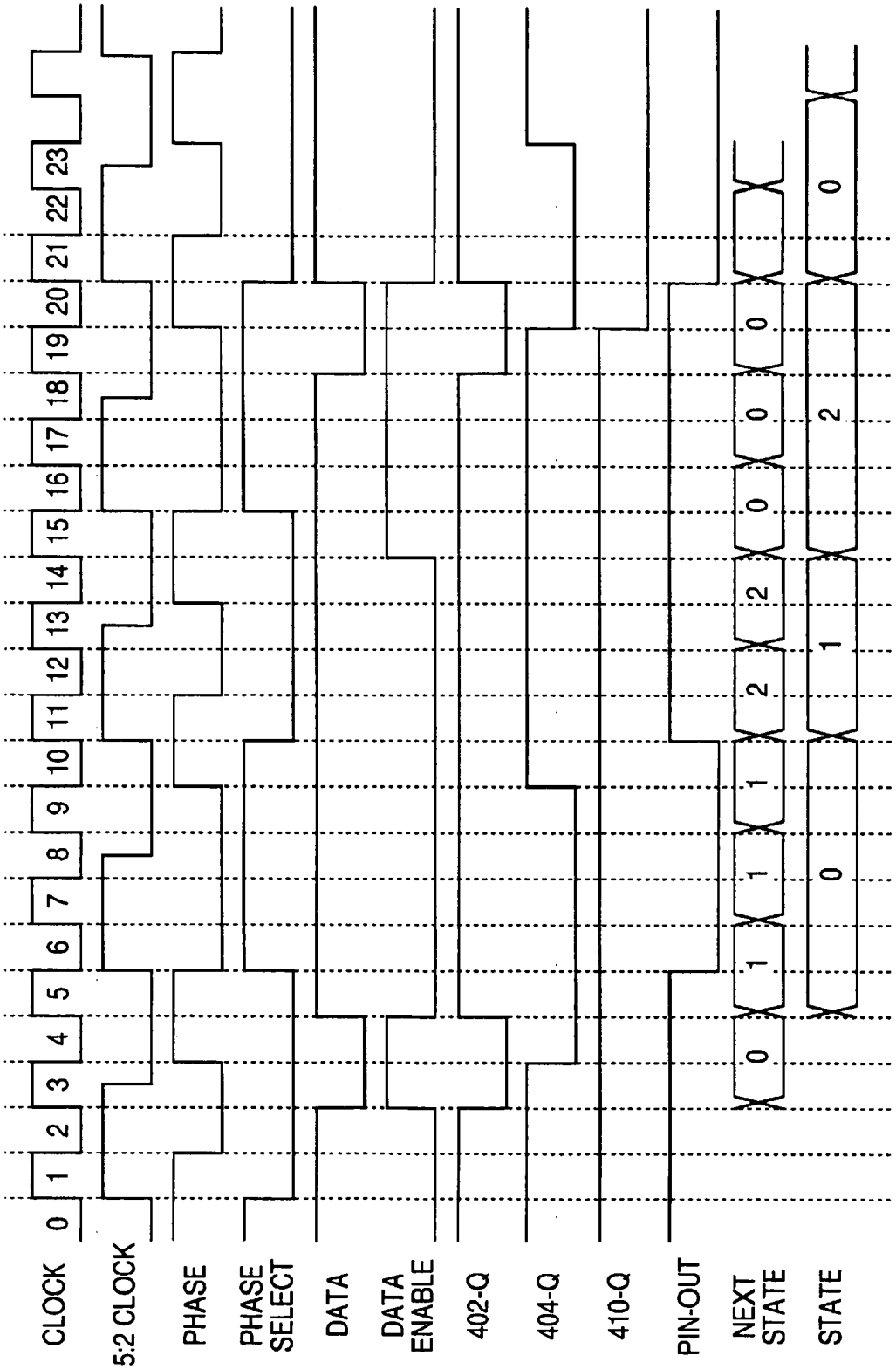

CLOCK
5:2 CLOCK
PHASE
PHASE SELECT
DATA
DATA ENABLE
402-Q
404-Q
410-Q
PIN-OUT
NEXT STATE
STATE
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
0 1 1 1 1 2 2 1 0
0 0 0 0 2 1 0

FIG. 6

LOW-LATENCY CIRCUIT FOR SYNCHRONIZING DATA TRANSFERS BETWEEN CLOCK DOMAINS DERIVED FROM A COMMON CLOCK

CROSS-REFERENCE TO RELATED APPLICATION

The present invention is related to that disclosed in U.S. patent application Ser. No. 09/477,488, filed concurrently herewith, entitled ALOW LATENCY CLOCK DOMAIN SYNCHRONIZATION CIRCUIT AND METHOD OF OPERATION. The above application is commonly assigned to the assignee of the present invention. The disclosure of the related patent application is hereby incorporated by reference for all purposes as if fully set forth herein.

TECHNICAL FIELD OF THE INVENTION

The present invention is directed, in general, to microprocessors and, more specifically, to synchronization circuits for transferring data between two different clock domains controlled by a processing device.

BACKGROUND OF THE INVENTION

The ever-growing requirement for high performance computers demands that state-of-the-art microprocessors execute instructions in the minimum amount of time. Over the years, efforts to increase microprocessor speeds have followed different approaches, including increasing the speed of the clock that drives the processor and reducing the number of clock cycles required to perform a given instruction.

Microprocessor speeds may also be increased by reducing the number of gate delays incurred while executing an operation. Under this approach, the microprocessor is designed so that each data bit or control signal propagates through the smallest possible number of gates when performing an operation. Additionally, the propagation delay through each individual gate is also minimized in order to further reduce the end-to-end propagation delay associated with transmitting a control signal or a data bit during the execution of an instruction.

One area where it is important to minimize propagation delays occurs at the interface between clock domains. Conventional microprocessors contain many clock signals that are derived from a basic high-frequency core clock. The core clock signal may be divided down to produce clock signals that are related, for example, by an N:1 ratio or by an (N+2):1 ratio. For instance, dividing the core clock by two and dividing the core clock by four yields two clock signals that are in a 2:1 ratio. Similarly, dividing the core clock by two and dividing the core clock by five yields two clock signals that are in a 2.5:1 ratio. These different clock domain signals may drive internal microprocessor components or may be brought off-chip to drive external devices, such as main memory, input/output (I/O) buses, and the like.

At the interface between two clock domains, there is no guarantee that a signal transmitted from a first clock domain will be synchronized with the clock in a second clock domain. Normally, synchronization between different clock domains is handled by a set of synchronizing flip-flops. A signal in a first clock domain is first registered in a flip-flop in the first clock domain. The output of that first flip-flop is then Adouble sampled@ by two flip-flop in the second clock domain. Double sampling means that the output of the first flip-flop feeds the input of a second flip-flop clocked in the second clock domain. The output of the second flip-flop feeds the input of a third flip-flop that also is clocked in the second clock domain. The output of this third flip-flop is properly synchronized with the second clock domain. An identical three flip-flop interface circuit is used to synchronize signals that are being transmitted in the reverse direction (i.e., from the second clock domain to the first clock domain). This synchronizing circuit, along with grey code encoding of multi-bit signals provides a means for synchronizing two asynchronous clock domains.

The chief drawback of the above-described flip-flop interface circuit is the fact that there are three gate propagation delays involved in transmitting a signal from one clock domain to another clock domain. This necessarily slows down the operation of the microprocessor and/or an external device communicating wit the microprocessor, since the circuits in the receiving domain receive the transmitted signal only after at least three propagation delays.

Therefore, there is a need in the art for improved microprocessor designs that maximize the throughput of a processor and any external devices communicating with the processor. In particular, there is a need in the art for improved circuits that interface signals between different clock domains. More particularly, there is a need for interface circuits that minimize the number of gate delays that affect a signal being transmitted from a faster clock domain to a slower clock domain, and vice versa.

SUMMARY OF THE INVENTION

The limitations inherent in the prior art described above are overcome by the present invention, which provides an interface circuit for synchronizing the transfer of data through an output port from a first clock domain driven by a first clock signal to a second clock domain driven by a second clock signal. In an advantageous embodiment, the interface circuit comprises 1) a first latch having a data input for receiving a data signal from the first clock domain, and enable input for receiving an enabling signal, a clock input for receiving the first clock signal, and an output; 2) a second latch having a data input coupled to the first latch output, a clock input for receiving a gating signal, a clock input for receiving the first clock signal, and an output; 3) a third latch having a data input for receiving the data signal, and enable input for receiving a phase sel3ect signal, a clock input for receiving the first clock signal, and an output; and 4) a multiplexer having a first data input coupled to the second latch output, a second data input coupled to the third latch output, and a selector input for selecting one of the first data input and the second data input for transfer to an output of the multiplexer.

According to one embodiment of the present invention, the second clock signal and the first clock signal are derived from a common core clock.

According to another embodiment of the present invention, a frequency of the second clock signal and a frequency of the first clock signal are in a ratio of N:1 where N is an integer.

According to still another embodiment of the present invention, a selection signal applied to the selector input selects the first data input of the multiplexer when a rising edge of the first clock signal is approximately in phase with a rising edge of the second clock signal.

According to yet another embodiment of the present invention, a frequency of the second clock signal and a frequency of the first clock signal are in a ratio of (N+2):1 where N is an integer.

According to a further embodiment of the present invention, a selection signal applied to the selector input selects the first data input of the multiplexer during one clock cycle of the second clock signal.

The present invention may also be embodied as an interface circuit for synchronizing the transfer of data from an output of a state machine in a first clock domain driven by a first clock signal to a second clock domain driven by a second clock signal. In an advantageous embodiment, the state machine interface circuit comprises 1) a first latch having a data input for receiving the state machine output, a clock input for receiving the first clock signal, and an output; and 2) a second latch having a data input coupled to the first latch output, a clock input for receiving a gating signal, and an output coupled to an input of the state machine.

According to one state machine interface embodiment of the present invention, the second clock signal and the first clock signal are derived from a common core clock.

According to another state machine interface embodiment of the present invention, a frequency of the second clock signal and a frequency of the first clock signal are in a ratio of N:1 where N is an integer.

According to still another state machine interface embodiment of the present invention, a frequency of the second clock signal and a frequency of the first clock signal are in a ratio of (N+2):1 where N is an integer.

The foregoing has outlined rather broadly the features and technical advantages of the present invention so that those skilled in the art may better understand the detailed description of the invention that follows. Additional features and advantages of the invention will be described hereinafter that form the subject of the claims of the invention. Those skilled in the art should appreciate that they may readily use the conception and the specific embodiment disclosed as a basis for modifying or designing other structures for carrying out the same purposes of the present invention. Those skilled in the art should also realize that such equivalent constructions do not depart from the spirit and scope of the invention in its broadest form.

Before undertaking the DETAILED DESCRIPTION, it may be advantageous to set forth definitions of certain words and phrases used throughout this patent document: the terms Ainclude@ and Acomprise,@ as well as derivatives thereof, mean inclusion without limitation; the term Aor,@ is inclusive, meaning and/or; the phrases Aassociated with@ and Aassociated therewith,@ as well as derivatives thereof, may mean to include, be included within, interconnect with, contain, be contained within, connect to or with, couple to or with, be communicable with, cooperate with, interleave, juxtapose, be proximate to, be bound to or with, have, have a property of, or the like; and the term Acontroller@ means any device, system or part thereof that controls at least one operation, such a device may be implemented in hardware, firmware or software, or some combination of at least two of the same. It should be noted that the functionality associated with any particular controller may be centralized or distributed, whether locally or remotely. Definitions for certain words and phrases are provided throughout this patent document, those of ordinary skill in the art should understand that in many, if not most instances, such definitions apply to prior, as well as future uses of such defined words and phrases.

BRIEF DESCRIPTION OF THE DRAWINGS

For a more complete understanding of the present invention, reference is now made to the following descriptions taken in conjunction with the accompanying drawings, in which:

FIG. **1** is a block diagram of an exemplary integrated processor system, including an integrated microprocessor in accordance with the principles of the present invention;

FIG. **2** illustrates in more detail the exemplary integrated microprocessor in FIG. **1** in accordance with one embodiment of the present invention;

FIG. **3** is a schematic diagram of a synchronization circuit for synchronizing the output of a state machine to a clock domain;

FIG. **4** is a schematic diagram of a synchronization circuit for synchronizing the transfer of data between two asynchronous clock domains;

FIG. **5** is a timing diagram illustrating the operations of the synchronization circuits illustrated in FIGS. **3** and **4** in accordance with an exemplary embodiment of the present invention; and

FIG. **6** is a timing diagram illustrating the operations of the synchronization circuits illustrated in FIGS. **3** and **4** in accordance with an exemplary embodiment of the present invention.

DETAILED DESCRIPTION

FIGS. **1** through **6**, discussed below, and the various embodiments used to describe the principles of the present invention in this patent document are by way of illustration only and should not be construed in any way to limit the scope of the invention. Those skilled in the art will understand that the principles of the present invention may be implemented in any suitably arranged integrated microprocessor.

Integrated Processor System

FIG. **1** is a block diagram of an exemplary integrated processor system, including integrated processor **100** in accordance with the principles of the present invention. Integrated microprocessor **100** includes central processing unit (CPU) **110**, which has dual integer and dual floating point execution units, separate load/store and branch units, and L1 instruction and data caches. Integrated onto the microprocessor die is graphics unit **120**, system memory controller **130**, and L2 cache **140**, which is shared by CPU **110** and graphics unit **120**. Bus interface unit **150** interfaces CPU **110**, graphics unit **120**, and L2 cache **140** to memory controller **130**.

Integrated memory controller **130** bridges processor **100** to system memory **160**, and may provide data compression and/or decompression to reduce bus traffic over external memory bus **165** which preferably, although not exclusively, has a RAMbusJ, fast SDRAM or other type protocol. Integrated graphics unit **120** provides TFT, DSTN, RGB, and other types of video output to drive display **180**.

Bus interface unit **150** interfaces, through I/O interface **152**, processor **100** to chipset bridge **190** for conventional peripheral bus **192** connection (e.g., PCI connection) to peripherals, such as sound card **194**, LAN controller **195**, and disk drive **196**, as well as fast serial link **198** (e.g., IEEE 1394 "firewire" bus and/or universal serial bus "USB") and relatively slow I/O port **199** for peripherals, such as a keyboard and/or a mouse. Alternatively, chipset bridge **160** may integrate local bus functions such as sound, disk drive control, modem, network adapter, etc.

Integrated CPU

FIG. **2** illustrates in more detail the exemplary integrated processor **100**, including CPU **110**, which is integrated with graphics controller **120**, memory controller **130**, and L2 unified cache **140** (e.g., 256 KB in size). CPU **110** includes an execution pipeline with instruction decode/dispatch logic **200** and functional units **250**.

Instruction decode/dispatch logic **200** decodes variable length x86 instructions into nodes (operations) each containing source, destination, and control logic. Each instruction maps into one or more nodes, which are formed into checkpoints for issue in parallel to functional units **250**. The exemplary execution pipeline includes dual integer units (EX) **255**, dual pipelined floating point units (FP) **260**, load/store unit (LDST) **265**, and branch unit (BR) **270**. Hence, a single checkpoint can include up to 2 EX, 2 FP, 1 LDST, and 1 BR nodes which can be issued in parallel. L1 data cache (DC) **280** (e.g., 16 KB in size) receives data requests from the LDST unit and, in the case of an L1 hit, supplies the requested data to appropriate EX or FP unit.

BR unit **270** executes branch operations based on flag results from the EX units. Predicted (taken/not-taken) and not-predicted (undetected) branches are resolved (mispredictions incur, for example, a 12 clock penalty) and branch information is supplied to BTB **275**, including branch address, target address, and resolution (taken or not taken). BTB **275** includes a 1 KB target cache, a 7-bit history and prediction ROM, and a 16-entry return stack.

Instruction decode/dispatch logic **200** includes L1 instruction cache (IC) **210** (e.g., 16 KB in size) which stores 32-byte cache lines (8 dwords/4 qwords). Each fetch operation, fetch unit **215** fetches a cache line of 32 instruction bytes from the L1 instruction cache to aligner logic **220**. Fetch unit **215** either (a) generates a fetch address by incrementing the previous fetch address (sequential fetch) or, (b) if the previous fetch address hit in BTB **275**, switches the code stream by supplying the fetch address for the cache line containing the target address provided by BTB **275**. Fetch unit **215** supplies a linear address simultaneously to L1 instruction cache **210** and BTB **275**. A two-level translation look-aside buffer (TLB) structure (a 32-entry L1 instruction TLB and a 256-entry shared L2 TLB) supplies a corresponding physical address to the L1 cache to complete cache access.

Aligner logic **220** identifies up to two x86 variable length instructions per clock. Instructions are buffered in instruction buffer **225**, along with decode and issue constraints. Decoder **230** transfers instructions from the instruction buffer to the appropriate one (as determined by decode constraints stored with the instruction) of decoders **D0**, **D1**, and Useq (a microsequencer). D0 and D1 define two decode slots (or paths) **S0** and **S1**, with the Useq decoder feeding nodes into both slots simultaneously.

D0 and D1 each decode single node EX/FPU/BR instructions that do not involve memory references (e.g., register-register integer and floating point operations and branch operations), while memory reference instructions, which decode into separate EX/FP and LDST nodes (e.g., register-memory integer and floating point operations), are constrained to D0. The Useq decoder handles instructions that decode into more than two nodes/operations (e.g., far calls/returns, irets, segment register loads, floating point divides, floating point transcendentals). Each such sequence of nodes are organized into one or more separate checkpoints issued in order to the functional units. Renaming logic **235** (including a logical-to-physical map table) renames sources and destinations for each node, mapping logical to physical registers.

Issue logic **240** organizes the renamed nodes from each slot into checkpoints that are scheduled for issue in order to the functional units. Most instructions can be dual issued with the nodes for each in the same checkpoint. Up to 16 checkpoints may be active (i.e., issued to functional units). Nodes are issued into reservation stations in each functional unit. Once in the reservation stations, the nodes complete execution out-of-order.

The dual EX0/EX1 (integer) units **255** are pipelined with separate copies of a physical register file, and execute and forward results in a single cycle. The dual FPU0/FPU1 units **260** include dual execution units (with separate FP physical register files) that support MMX and 3DNow instructions, as well as standard x87 floating point, instruction execution. FPU0 includes a pipelined FAdder and FPU1 includes a pipelined Fmultipler, both supporting packed SIMD operations.

Integer multiply operations are issued to FPU1 with the Fmultiplier, and integer divide operations are issued as separate nodes to both FPU0 and FPU1, so that integer EX operations can execute in parallel with integer multiplies and divides. Results are forwarded between EX0/EX1 and FPU0/FPU1 in a single cycle.

LDST unit **265** executes memory reference operations as loads/stores to/from data cache **280** (or L2 cache **140**). LDST unit **265** performs pipelined linear address calculation and physical (paged) address translation, followed by data cache access with the physical (translated) address. Address translations are performed in order using a two-level TLB structure (a 32 entry L1 data TLB and the 256 entry shared L2 TLB). Up to four pending L1 misses can be outstanding. Missed data returns out of order (from either L2 cache **140** or system memory **160**).

Exemplary 16 KB L1 instruction cache **210** is single-ported 4-way associative, with 2 pending misses. Exemplary 16 KB L1 data cache **280** is non-blocking, dual-ported (one load port and one store/fill port), 4-way associative, with 4 pending misses. Both L1 caches are indexed with the linear address and physically tagged with the TLB (translated) address. In response to L1 misses, L2 cache **140** transfers an entire cache line (32 bytes/256 bits) in one cycle with a 7 clock access latency for L1 misses that hit in L2 cache **140**.

Exemplary 256 KB L2 cache **140** is 8-way associative and 8-way interleaved. Each interleave supports one L1 (code/data) miss per cycle, and either one L1 store or one L2 fill per cycle. Portions or all of 2 of the 8 ways may be locked down for use by graphics controller **120**.

For integer register-to-register operations, the execution pipeline is eleven (11) stages from code fetch to completion: two cache access stages (IC1 and IC2), two alignment stages (AL1 and AL2), three decode/rename stages (DEC0–DEC2), checkpoint issue stage (ISS), and reservation stage (RS), followed by the execute and result write-back/forward stages (EX and WB). For integer register-memory operations, the LDST unit pipeline adds an additional four stages between RS and EX: address calculation (AC), translation (XL), and data cache access and drive back DC and DB. The floating point adder pipeline comprises four stages and the floating point multiply pipeline comprises five stages.

Different functional blocks in integrated processor **100** may operate at different clock speeds. Each group of circuits that are driven at a specified clock speed is referred to as a clock domain. As described above in the Background, special synchronization circuitry is needed to transfer data from one clock domain to another clock domain. However, because all of the clock domains in integrated processor **100** are derived from a common core clock, the phase and frequency relationships between the different clock domains are known. The present invention use knowledge of the phase and frequency relationships between clock domains to provide unique synchronization circuits that minimize the number of gates and clock delays encountered when transferring data from one domain to another domain.

FIG. **3** is a schematic diagram of exemplary synchronization circuit **300** for synchronizing the output of a state machine to a clock domain. Exemplary synchronization circuit **300** comprises latch **302**, latch **304**, inverter **306**, inverter **307**, AND gate **308**, and state machine logic circuit **310**. The data input (D) of latch **302** is connected to the Anext state@ output (NEXT) of state machine logic circuit **310**, and the enable input (EN) of latch **302** is permanently connected to a Logic 1 enabling signal. Latch **302** transfers NEXT to its Q output on the rising edge of CLK.

The output of latch **302** is connected to the data (D) input of latch **304**. Inverters **306** and **307** invert the CLK signal. The inverted CLK signal is one input to AND gate **308**. The other input of AND gate **308** receives the PHASE signal. The output of AND gate **308** is a gated clock signal that is Ahigh@ (or Logic 1) when inverted CLK and PHASE are both high. The output of AND gate **308** is connected to the enable (EN) input of latch **304**. Latch **304** transfers the clocked output of latch **302** to the Q output of latch **304** on the rising edge of the inverted CLK signal from inverter **307**, providing an output which is synchronized with clock domain of the CLK domain. The Q output of latch **304** represents the current state (CURRENT) which is connected as the input to Logic circuit **310**. Since the CURRENT input to state machine logic circuit **310** is synchronized with the CLK signal, the NEXT output of state machine logic circuit **310** is also synchronized with the CLK signal.

FIG. **4** is a schematic diagram of exemplary synchronization circuit **400** for synchronizing the transfer of data between two asynchronous clock domains. Synchronization circuit **400** transfers the DATA signal off-chip to another circuit connected to pin **430**. Latches **402**, **404**, and **410** and multiplexer **412** form synchronizing circuit for an input data signal, labeled ADATA@ in FIG. **4**. Latches **420**, **422**, and **424** and multiplexer **426** form a synchronizing circuit for an input data enable signal, labeled ADATA ENABLE@ in FIG. **4**. Inverter **406** and AND gate **408** provide a gated inverted clock signal for use by both synchronizing circuit groups. Inverter **428** and tri-state driver **414** provide means for transferring synchronized data during the high level of the DATA ENABLE signal from multiplexer **412** to pin **430**.

Latch **402** transfers the DATA signal from input D to output Q on the rising edge of the CLK signal. The enable (EN) input to latch **402** is connected to Logic 1. The output Q of latch **402** is connected to input D of latch **404**. Inverter **406** inverts CLK and supplies inverted CLK as an input to AND gate **408**. The other input of AND gate **408** receives the signal labeled APHASE@ in FIG. **4**. The inverted CLK output from AND gate **408** is supplied as the enable (EN) input for latches **404**, **410**, **422**, and **424**.

Inverter **407** inverts the CLK signal and clocks latch **404**. Latch **404** transfers the output of latch **402** to its output Q on the rising edge of the output from inverter **407**. In a similar manner, latch **410** transfers the DATA signal from its D input to its Q output on the rising edge of the output of inverter **407**. The output of latches **404** and **410** are provided as data inputs to multiplexer **412**. The phase-select signal, labeled APHASE SELECT@ in FIG. **4** selects one of the two data inputs of multiplexers **412** and **426**. Thus, multiplexer **412** transfers the output of latch **404** to its output when PHASE SELECT is high and multiplexer **412** transfers the output of latch **410** to its output when PHASE SELECT is low.

The output of multiplexer **412** is connected to the non-inverting input of tri-state driver **414**. Inverter **428** inverts the output from multiplexer **426** and provides this as the inverted input to tri-state driver **414**. Tri-state driver **414** transfers the output of multiplexer **412** to its output when the output of inverter **428** is low (Logic 0). Thus, tri-state driver **414** transfers the output of multiplexer **412** to pin **430** when the output of multiplexer **426** is high. Otherwise, the tri-state driver **414** provides a high impedance to pin **430**.

As previously described, the synchronizing circuit composed of latches **420**, **422**, and **424**, and multiplexer **426** operates in the same manner as previously described for the DATA signal, except that the DATA ENABLE signal is transferred in place of the DATA signal. The Q outputs of latches **422** and **424** are provided as inputs to multiplexer **426**, with the PHASE SELECT signal controlling the output of multiplexer **426**. Multiplexer **426** transfers the output of latch **422** to inverter **428** when PHASE SELECT is high and transfers the output of latch **424** to inverter **428** when PHASE SELECT is low. As previously discussed, tri-state driver **414** provide means for transferring the DATA signal from multiplexer **412** to pin **430** during the high level of DATA ENABLE signal from multiplexer **426**.

FIG. **5** is a timing diagram illustrating the operations of the synchronization circuits illustrated in FIGS. **3** and **4** in accordance with an exemplary embodiment of the present invention. The timing diagram shows the signals: CLOCK (labeled ACLK@ in FIGS. **3** and **4**), 2:1 CLOCK, PHASE, PHASE SELECT, DATA, DATA ENABLE, PIN-OUT, NEXT STATE, and STATE (labeled ACURRENT@ in FIG. **3**).

CLOCK is square wave in which high and low intervals (or pulses) are sequentially numbered. Even numbers represent the low pulses of CLOCK and odd numbers represent the high pulses of CLOCK. An even and odd numbered pair of adjacent pulses represents a single cycle for CLOCK. The 2:1 CLOCK time line represents a clock signal which is running at half the rate of CLOCK. For this example, 2:1 CLOCK transitions to high or low when CLOCK transitions from low to high. The time line for PHASE depicts an inverse relationship to the 2:1 CLOCK time line (i.e., high when 2:1 CLOCK is low and low when 2:1 CLOCK is high). For the purposes of this example, PHASE SELECT is shown as always high.

The DATA signal is only transferred to the output of multiplexer **412** when the PHASE signal is high. During pulses **3** and **4** (i.e., one cycle of CLOCK), the DATA signal goes low when PHASE is high. At the same time, during pulses **3** and **4** (i.e., one cycle of CLOCK), the DATA ENABLE signal goes high and is clocked through to tri-state driver **414**. Thus, the DATA signal is driven through to PIN-OUT which goes from high to low. Subsequently, during pulses **5** through **12**, the DATA signal goes high again. However, the PHASE does not go high again until pulses **7** and **8**. During pulses **7** and **8**, the high DATA signal is driven through latch **404** and multiplexer **412** to tri-state driver **414**. Since DATA ENABLE signal is still held high by latch **422**, tri-state driver **414** is still enabled. Thus, the DATA signal is driven through to PIN-OUT, which goes from low to high. Another exemplary pulse of the DATA signal is driven through to PIN OUT during pulses **17–20**.

In FIG. **3**, the output of latch **304**, labeled ACURRENT@ in FIG. **3** and ASTATE@ in FIG. **5**, can only change when PHASE is high and CLOCK is low (i.e., pulses **4**, **8**, **12**, **16**, etc.). Thus, STATE transitions to State 0 during pulse **4**, to State 1 during pulse **8**, to State 2 during pulse **12**, and finally back to State 0 during pulse **16**.

FIG. **6** is a timing diagram illustrating the operations of the synchronization circuits illustrated in FIGS. **3** and **4** in accordance with an exemplary embodiment of the present invention. For this example, CLOCK is 2.5 times faster than 5:2 CLOCK, with the positive transition of 5:2 CLOCK

coinciding with the beginning of every fifth half-cycle of CLOCK. The PHASE signal=s high interval always begins and ends with a falling edge of CLOCK and it remains high for one CLOCK cycle. PHASE SELECT essentially represents a 5:1 CLOCK which makes its transitions on the rising edge of the 5:2 CLOCK. In other words, PHASE SELECT cycles at half the rate of 5:2 CLOCK and one fifth the rate of CLOCK.

As in FIG. **5**, the DATA signal is only transferred to the output of multiplexer **412** when the PHASE signal is high. Latches **404** and **410** are clocked and transfer data from input to output when PHASE is high and CLOCK is low. Latches **422** and **424** are clocked by the inverted CLOCK and transfer data from input to output when PHASE SELECT is high. PHASE SELECT is used to select the output of multiplexers **412** and **426** so that the PIN OUT signal is synchronized to the domain of the 5:2 clock signal.

Although the present invention has been described in detail, those skilled in the art should understand that they can make various changes, substitutions and alterations herein without departing from the spirit and scope of the invention in its broadest form.

What is claimed is:

1. An interface circuit for synchronizing the transfer of data through an output port from a first clock domain driven by a first clock signal to a second clock domain driven by a second clock signal, the interface circuit comprising:

a first latch having a data input for receiving a data signal from said first clock domain, an enable input for receiving said first clock signal, a clock input for receiving said first clock signal; and an output;

a second latch having a data input coupled to said first latch output, a clock input for receiving a gating signal, a clock input for receiving said first clock signal, and an output;

a third latch having a data input for receiving said data signal, an enable input for receiving a phase select signal, a clock input for receiving said first clock signal, and an output; and

a multiplexer having a first data input coupled to said second latch output, a second data input coupled to said third latch output, and a selector input for selecting one of said first data input and said second data input for transfer to an output of said multiplexer.

2. The interface circuit set forth in claim **1** wherein said second clock signal and said first clock signal are derived from a common core clock.

3. The interface circuit set forth in claim **2** wherein a frequency of said second clock signal and a frequency of said first clock signal are in a ratio of N:1 where N is an integer.

4. The interface circuit set forth in claim **3** wherein a selection signal applied to the selector input selects said first data input of said multiplexer when a rising edge of said first clock signal is approximately in phase with a rising edge of said second clock signal.

5. The interface circuit set forth in claim **2** wherein a frequency of said second clock signal and a frequency of said first clock signal are in a ratio of (N+2):1 where N is an integer.

6. The interface circuit set forth in claim **5** wherein a selection signal applied to the selector input selects said first data input of said multiplexer during one clock cycle of said second clock signal.

7. An interface circuit for synchronizing the transfer of data from an output of a state machine in a first clock domain driven by a first clock signal to a second clock domain driven by a second clock signal, the interface circuit comprising:

a first latch having a data input for receiving said state machine output, an enable input that is set to an enabled value, and an output; and

a second latch having a data input coupled to said first latch output, an enable input for receiving a gating signal, a clock input for receiving said first clock signal, and an output coupled to an input of said state machine.

8. The interface circuit set forth in claim **7** wherein said second clock signal and said first clock signal are derived from a common core clock.

9. The interface circuit set forth in claim **8** wherein a frequency of said second clock signal and a frequency of said first clock signal are in a ratio of N:1 where N is an integer.

10. The interface circuit set forth in claim **8** wherein a frequency of said second clock signal and a frequency of said first clock signal are in a ratio of (N+2):1 where N is an integer.

11. A computer system comprising:

a pipelined, x86-compatible processor having dual integer and dual floating point execution units, separate load/store and branch units, an L1 instruction cache and an L1 data cache;

system memory for storing data or instructions;

a core clock; and

an interface circuit for synchronizing the transfer of data through an output port from a first clock domain driven by a first clock signal to a second clock domain driven by a second clock signal, the interface circuit comprising:

a first latch having a data input for receiving a data signal from said first clock domain, a clock input for receiving said first clock signal, an enable input that is set to an enabled value and an output;

a second latch having a data input coupled to said first latch output, an enable input for receiving a gating signal, a clock input for receiving said first clock signal, and an output;

a third latch having a data input for receiving said data signal, a enable input for receiving a phase select signal, a clock input for receiving said first clock signal, and an output; and

a multiplexer having a first data input coupled to said second latch output, a second data input coupled to said third latch output, and a selector input for selecting one of said first data input and said second data input for transfer to an output of said multiplexer.

12. The computer system set forth in claim **11** wherein said second clock signal and said first clock signal are derived from said core clock.

13. The computer system set forth in claim **12** wherein a frequency of said second clock signal and a frequency of said first clock signal are in a ratio of N:1 where N is an integer.

14. The computer system set forth in claim **13** wherein a selection signal applied to the selector input selects said first data input of said multiplexer when a rising edge of said first clock signal is approximately in phase with a rising edge of said second clock signal.

15. The computer system set forth in claim **12** wherein a frequency of said second clock signal and a frequency of said first clock signal are in a ratio of (N+2):1 where N is an integer.

16. The computer system set forth in claim **15** wherein a selection signal applied to the selector input selects said first

data input of said multiplexer during one clock cycle of said second clock signal.

17. A computer system comprising:

a pipelined, x86-compatible processor having dual integer and dual floating point execution units, separate load/store and branch units, an L1 instruction cache and an L1 data cache;

system memory for storing data or instructions;

a core clock; and

an interface circuit for synchronizing the transfer of data from an output of a state machine in a first clock domain driven by a first clock signal to a second clock domain driven by a second clock signal, the interface circuit comprising:

a first latch having a data input for receiving said state machine output, a clock input for receiving said first clock signal, an enable input set to an enabled value, and an output; and

a second latch having a data input coupled to said first latch output, an enable input for receiving a gating signal, a clock input for receiving said first clock signal, and an output coupled to an input of said state machine.

18. The computer system set forth in claim 17 wherein said second clock signal and said first clock signal are derived from a common core clock.

19. The computer system set forth in claim 18 wherein a frequency of said second clock signal and a frequency of said first clock signal are in a ratio of N:1 where N is an integer.

20. The computer system set forth in claim 18 wherein a frequency of said second clock signal and a frequency of said first clock signal are in a ratio of (N+2):1 where N is an integer.

* * * * *

US005522085A

United States Patent [19]

Harrison et al.

[11] Patent Number: 5,522,085

[45] Date of Patent: May 28, 1996

[54] **ARITHMETIC ENGINE WITH DUAL MULTIPLIER ACCUMULATOR DEVICES**

[75] Inventors: **Calvin W. Harrison**; **Susan L. Gilfeather**; **John B. Gehman, Jr.**, all of Scottsdale, Ariz.

[73] Assignee: **Motorola, Inc.**, Schaumburg, Ill.

[21] Appl. No.: **394,054**

[22] Filed: **Feb. 24, 1995**

Related U.S. Application Data

[63] Continuation of Ser. No. 170,145, Dec. 20, 1993.

[51] **Int. Cl.**[6] **G06F 7/52**

[52] **U.S. Cl.** **395/800**; 364/754; 364/736; 364/736.5; 364/768

[58] **Field of Search** 395/800; 364/DIG. 1, 364/726, 754, 724, 744, 736; 382/41

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,604,721	8/1986	Gray	364/726
4,768,159	8/1988	Gray et al.	364/726
4,802,111	1/1989	Barkan et al.	364/724.1
4,868,776	9/1989	Gray et al.	364/766
4,876,660	10/1989	Owen et al.	364/754
5,175,702	12/1992	Beraud et al.	364/736
5,220,525	6/1993	Anderson et al.	364/760
5,278,781	1/1994	Aono et al.	364/736
5,329,283	7/1994	Smith	342/25
5,379,351	1/1995	Fandrianto et al.	382/41

OTHER PUBLICATIONS

"LH9124 Digital Signal Processor" from SHARP Data Sheet, (Ref. Code SMT90033) Jul. 1992.

An article entitled "LH9320 DSP Address Generator" from SHARP Data Sheet, (FDS–Rev. A,Sep. 30, 1992).

An article entitled "LH9124 Digital Signal Processor" from SHARP Data Sheet, (Ref. Code SMT90033 Jul. 1992).

An article entitled "Frequency Domain Array Processor, A66540A/4K, 16K, 64K, A66540B/4K, 16K, 64K", from Preliminary User's Guide, Revision Mar. 1991 (Array Microsystems, 1420 Quail Lake Loop, Colorado Springs, CO 80906).

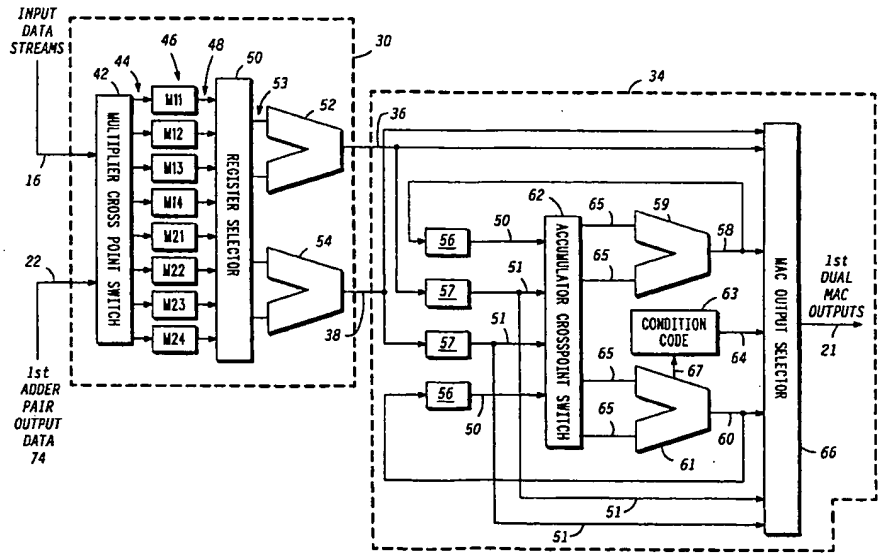
An article entitled "A 300–MOPS Video Signal Processor with a Parallel Architecture" by Toshihiro Minami, Ryota Kasai, Yamauchi, Yutaka Tashiro, Jun–ichi Takahashi and Shigeru Date, from IEEE Journal of Solid–State Circuits, vol. 26, No. 12, (Dec. 1991).

Primary Examiner—Alyssa H. Bowler
Assistant Examiner—Walter D. Davis, Jr.
Attorney, Agent, or Firm—Bradley J. Botsch, Sr.; Jeffrey D. Nehr

[57] **ABSTRACT**

An arithmetic engine includes a first dual multiplier accumulator (MAC) for receiving input data and for producing first dual MAC output data. A second dual MAC is coupled in parallel to the first dual MAC. The second dual MAC receives the input data and produces second dual MAC output data. An adder array is coupled to both the first dual MAC and to the second dual MAC. The adder array receives the input data, the first dual MAC output data, and the second dual MAC output data and produces arithmetic engine output data. Each dual MAC comprises a multiplier cross point switch, multiplier registers, a register selector, and parallel multipliers. Each adder array comprises a cross point switch, adder registers, a register selector, adder, and condition code determiner.

12 Claims, 5 Drawing Sheets



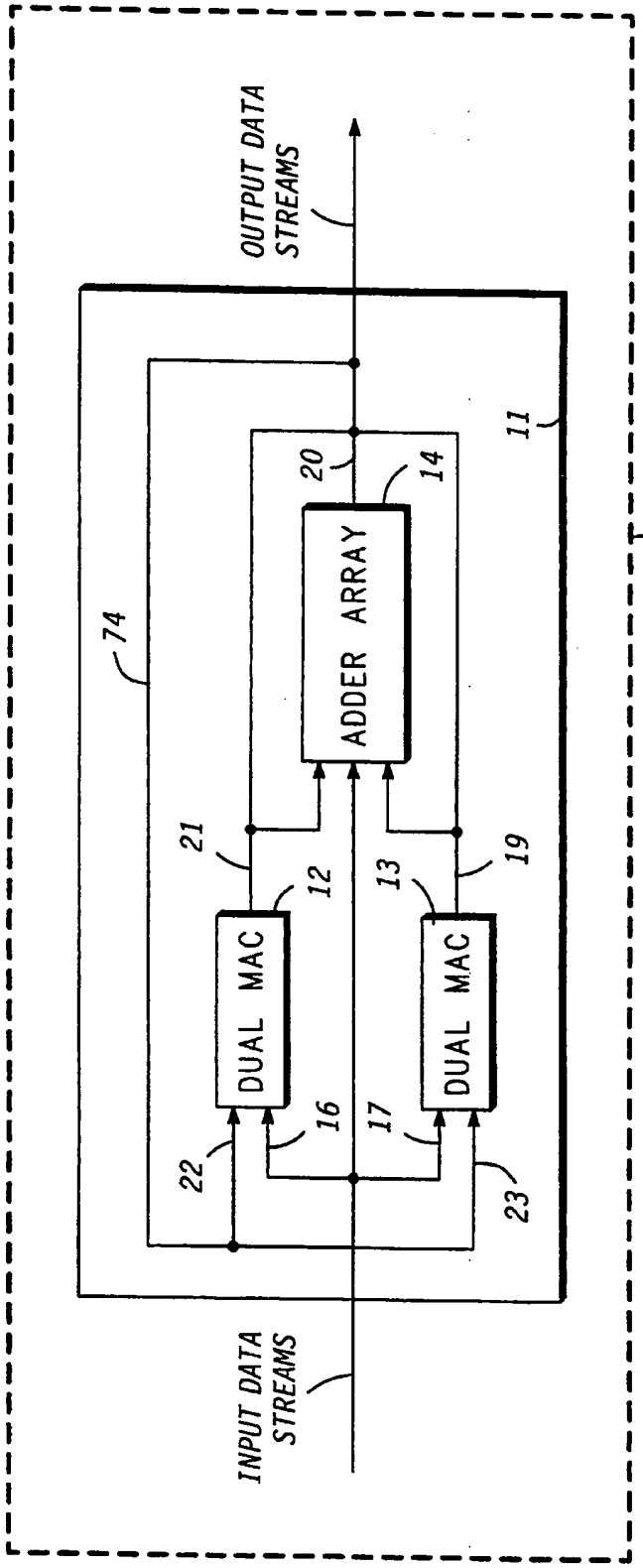

OUTPUT DATA STREAMS
ADDER ARRAY
DUAL MAC
DUAL MAC
INPUT DATA STREAMS
10
11
12
13
14
16
17
19
20
21
22
23
74

FIG. 1

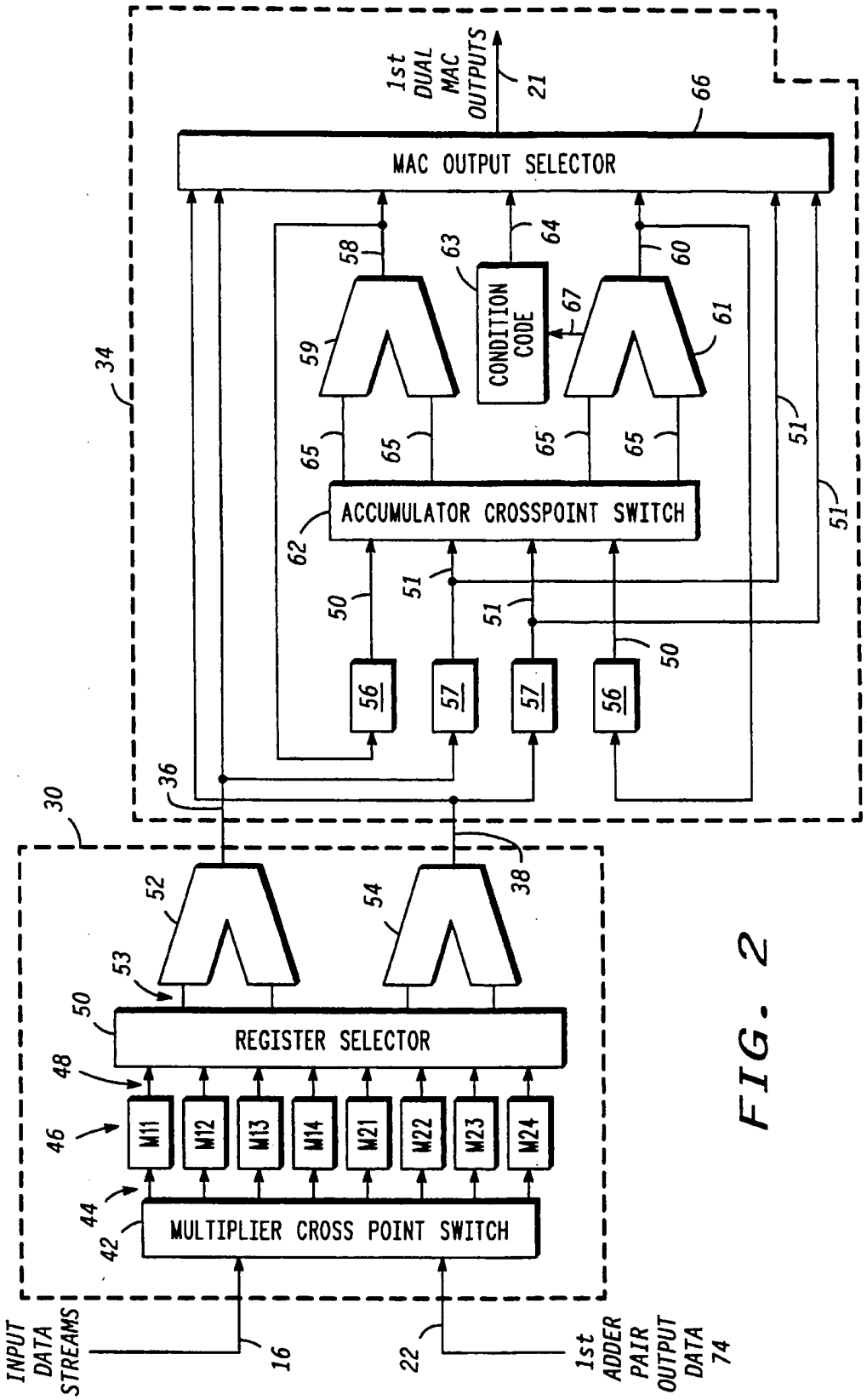

1st DUAL MAC OUTPUTS
21
66
MAC OUTPUT SELECTOR
58
63
64
60
CONDITION CODE
67
59
61
34
65
65
65
65
51
51
ACCUMULATOR CROSSPOINT SWITCH
62
50
51
51
50
56
57
57
56
36
30
38
52
54
53
50
REGISTER SELECTOR
48
46
W11
W12
W13
W14
W21
W22
W23
W24
44
42
MULTIPLIER CROSS POINT SWITCH
INPUT DATA STREAMS
16
22
1st ADDER PAIR OUTPUT DATA
74

FIG. 2

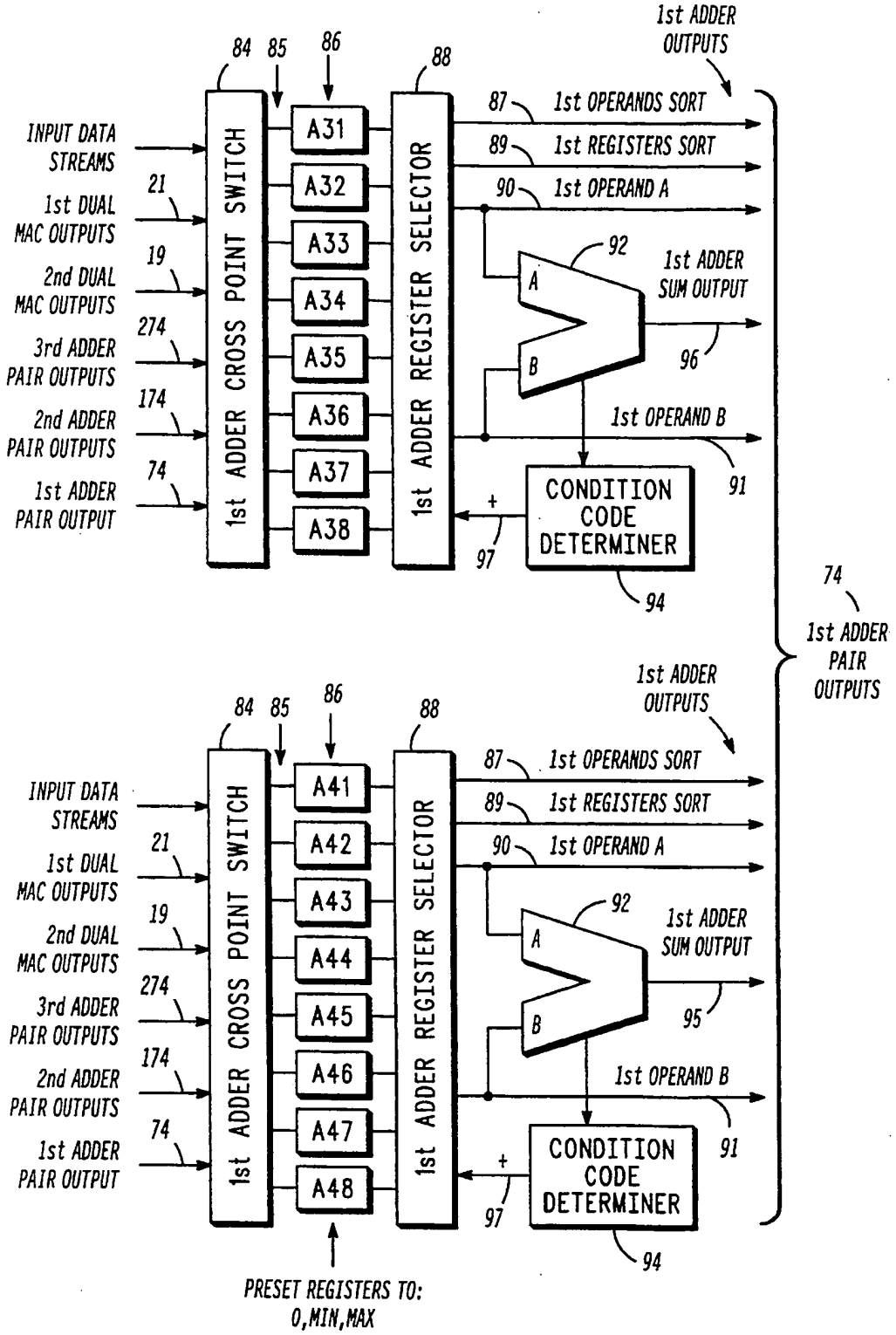

1st ADDER OUTPUTS
84
85
86
88
INPUT DATA STREAMS
21
1st DUAL MAC OUTPUTS
19
2nd DUAL MAC OUTPUTS
274
3rd ADDER PAIR OUTPUTS
174
2nd ADDER PAIR OUTPUTS
74
1st ADDER PAIR OUTPUT
1st ADDER CROSS POINT SWITCH
A31
A32
A33
A34
A35
A36
A37
A38
1st ADDER REGISTER SELECTOR
87
1st OPERANDS SORT
89
1st REGISTERS SORT
90
1st OPERAND A
92
A
B
1st ADDER SUM OUTPUT
96
1st OPERAND B
91
+
97
CONDITION CODE DETERMINER
94
74
1st ADDER PAIR OUTPUTS
1st ADDER OUTPUTS
84
85
86
88
INPUT DATA STREAMS
21
1st DUAL MAC OUTPUTS
19
2nd DUAL MAC OUTPUTS
274
3rd ADDER PAIR OUTPUTS
174
2nd ADDER PAIR OUTPUTS
74
1st ADDER PAIR OUTPUT
1st ADDER CROSS POINT SWITCH
A41
A42
A43
A44
A45
A46
A47
A48
1st ADDER REGISTER SELECTOR
87
1st OPERANDS SORT
89
1st REGISTERS SORT
90
1st OPERAND A
92
A
B
1st ADDER SUM OUTPUT
95
1st OPERAND B
91
+
97
CONDITION CODE DETERMINER
94
PRESET REGISTERS TO: 0,MIN,MAX

FIG. 3

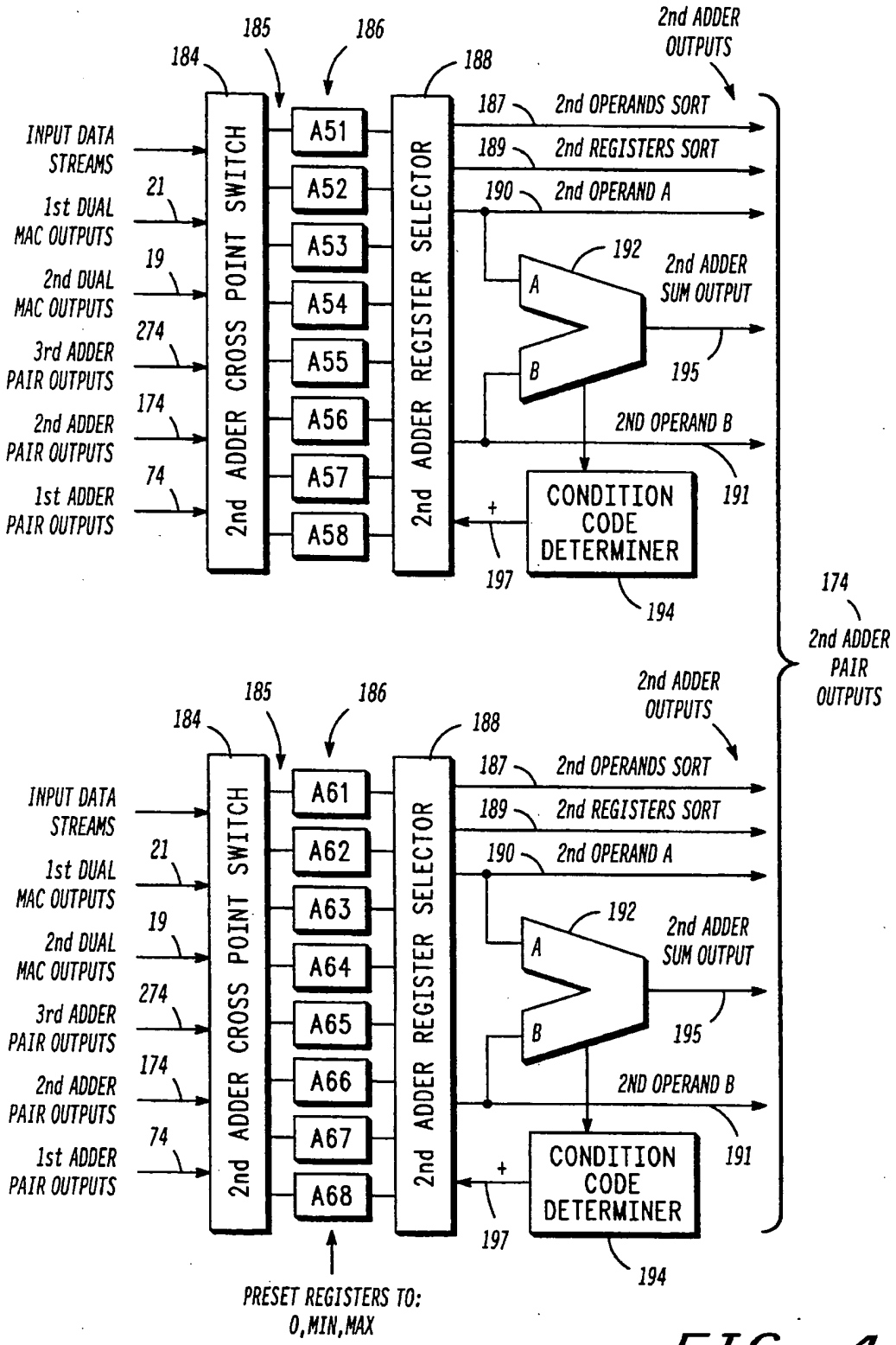

INPUT DATA STREAMS
1st DUAL MAC OUTPUTS
2nd DUAL MAC OUTPUTS
3rd ADDER PAIR OUTPUTS
2nd ADDER PAIR OUTPUTS
1st ADDER PAIR OUTPUTS
21
19
274
174
74
184
185
186
188
2nd ADDER CROSS POINT SWITCH
A51
A52
A53
A54
A55
A56
A57
A58
2nd ADDER REGISTER SELECTOR
2nd ADDER OUTPUTS
187 2nd OPERANDS SORT
189 2nd REGISTERS SORT
190 2nd OPERAND A
192
2nd ADDER SUM OUTPUT
195
2ND OPERAND B
191
CONDITION CODE DETERMINER
197
194
174
2nd ADDER PAIR OUTPUTS
A61
A62
A63
A64
A65
A66
A67
A68
PRESET REGISTERS TO: 0,MIN,MAX

FIG. 4

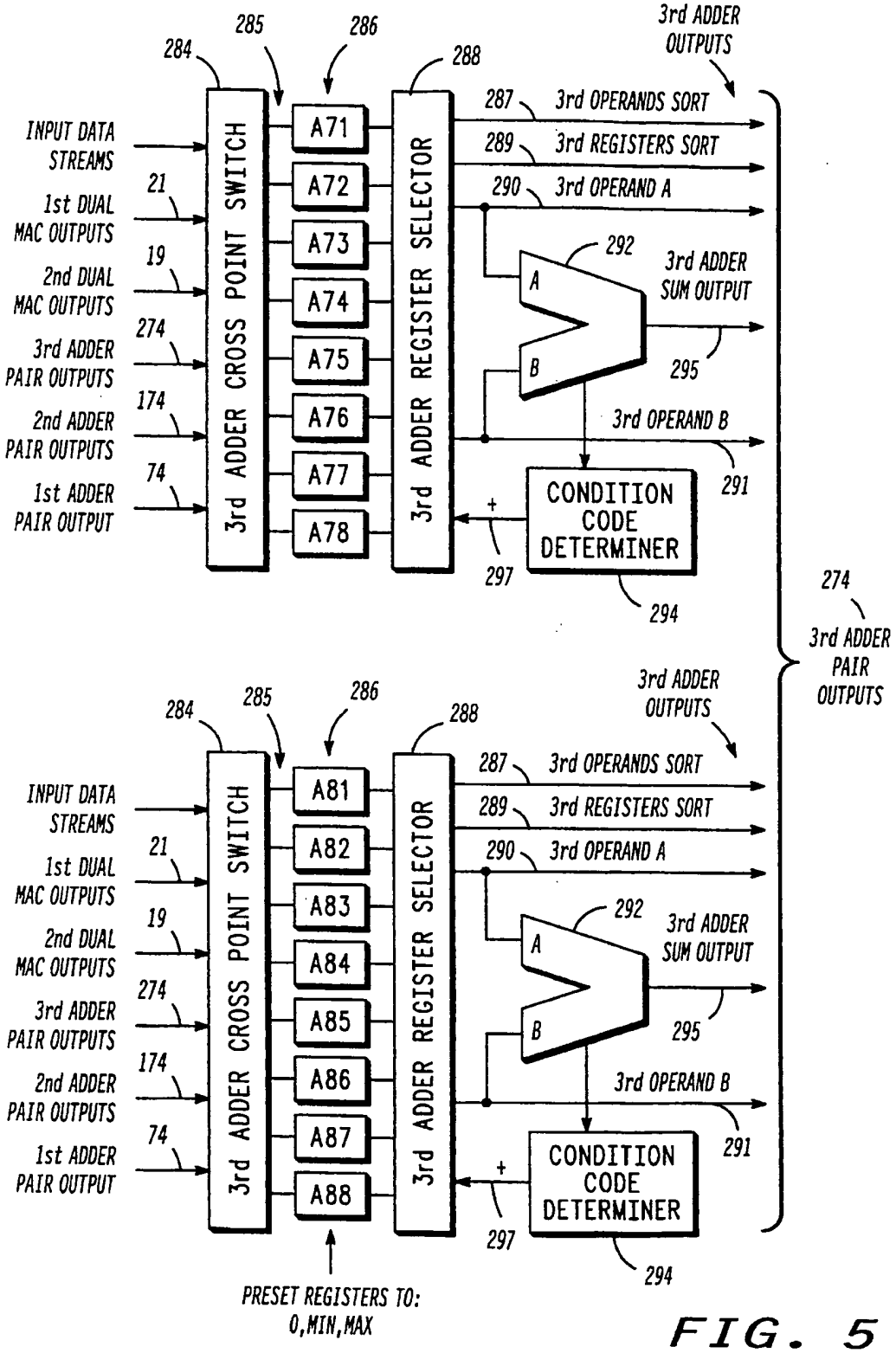

284
285
286
288
3rd ADDER OUTPUTS
INPUT DATA STREAMS
1st DUAL MAC OUTPUTS
21
2nd DUAL MAC OUTPUTS
19
3rd ADDER PAIR OUTPUTS
274
2nd ADDER PAIR OUTPUTS
174
1st ADDER PAIR OUTPUT
74
3rd ADDER CROSS POINT SWITCH
A71
A72
A73
A74
A75
A76
A77
A78
3rd ADDER REGISTER SELECTOR
287 3rd OPERANDS SORT
289 3rd REGISTERS SORT
290 3rd OPERAND A
292
A
B
3rd ADDER SUM OUTPUT
295
3rd OPERAND B
291
+
297
CONDITION CODE DETERMINER
294
274
3rd ADDER PAIR OUTPUTS
A81
A82
A83
A84
A85
A86
A87
A88
PRESET REGISTERS TO: 0,MIN,MAX

FIG. 5

ARITHMETIC ENGINE WITH DUAL MULTIPLIER ACCUMULATOR DEVICES

This application is a continuation of prior application Ser. No. 08/170,145, filed Dec. 20, 1993.

FIELD OF THE INVENTION

This invention relates in general to digital signal processing and in particular to arithmetic engines for such applications.

BACKGROUND OF THE INVENTION

High speed digital signal processing (DSP) in communication system applications require low power implementations of high performance DSP capabilities, especially for hand held devices. These DSPs must be capable of efficiently performing the spectral analysis and digital filtering required for hand-held spread spectrum communication equipment, high speed modems and all digital radios. These applications require matrix math operations, complex arithmetic operations, Fast Fourier Transform (FFT) calculations, encoding/decoding, searching and sorting. Optimization requires that the DSP arithmetic engine use many resources efficiently. For example, for execution of the complex multiply operation:

$$(x1+iy1)(x2+iy2)=(x1x2-y1y2)+i(x1y2+y1x2)$$

four real multiplications and two real additions are required to perform this operation in one instruction. More multipliers and adders would not improve the speed, less would require more instructions.

What is needed is a high performance arithmetic engine that can be optimized to perform higher radix FFTs, complex multiplication and high speed data sorting. Higher radix FFTs require less memory accesses because intermediate results are stored in registers in the arithmetic engine. The throughput is therefore faster because the arithmetic engine can be pipelined. A radix 8 4096 point FFT requires ⅔ less data fetches than a radix 2 FFT and ⅓ less than a radix 4. The complex multiply is the basis for most DSP algorithms and is therefore an important performance target. Sorting is required to do statistical filtering and interpret results. Sorting can determine the location of peaks, depressions and statistically variant data which are typical signal analysis objectives. Such an arithmetic engine requires many resources which must be used efficiently because of size and power limitations.

Typical arithmetic engines for general purpose DSPs include one multiplier and one adder referred to as a multiplier-accumulator (MAC). Arithmetic engines of DSPs that are more application specific can contain an array of several multipliers and adders. The arithmetic engines of the latter DSPs are typically optimized to perform a Radix-4 butterfly and cannot do high speed sorting.

Typical arithmetic engines are also not capable of efficiently handling the arithmetic computations required for all digital communications applications. General purpose DSPs are capable of performing Radix 2 butterflies efficiently but can not do higher radix butterflies because they do not have sufficient resources. The FFT requires many complex multiplications, and each complex multiply requires four real multiplications and two real additions. Since general purpose DSPs have only one MAC, it takes four passes through the arithmetic engine of such a DSP to perform just one complex multiply. Engines which have more than one MAC usually limit the resources to Radix 4 butterfly operations.

Typical arithmetic engines are also not capable of handling high speed data sorting. A typical data sort requires comparison of two pieces of data which results in a condition code. The condition code is passed to the execution unit which then causes the program to branch in one of two paths on a subsequent instruction. The next instruction after the branch moves the data to its new location based on the results of the data comparison. This method requires a minimum of three instructions and can only operate on one data pair at a time. Typical arithmetic engines do not have sufficient data path switching to perform hardware sorts on multiple data pairs.

SUMMARY OF THE INVENTION

Accordingly, it is an advantage of the present invention to provide a new and improved arithmetic engine. It is an additional advantage that the arithmetic engine operates on complex data in ordered pairs and is optimized for Radix 8 processing. It is still a further advantage of the invention that the arithmetic engine provides high speed data sorting, using full computing resources by recirculating data within the engine by feeding outputs back to inputs.

To achieve these advantages, an arithmetic engine is contemplated which includes a first dual multiplier accumulator (MAC) for receiving input data at a first dual MAC first input and for producing first dual MAC output data at a first dual MAC output. A second dual MAC is coupled in parallel to the first dual MAC. The second dual MAC receives the input data at a second dual MAC first input and produces second dual MAC output data at a second dual MAC output. An adder array is coupled to both the first dual MAC and to the second dual MAC. The adder array is for receiving the input data, the first dual MAC output data, and the second dual MAC output data and for producing arithmetic engine output data at an adder array output.

The above and other features and advantages of the present invention will be better understood from the following detailed description taken in conjunction with the accompanying drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

In FIG. **1**, there is shown a schematic of an arithmetic engine in a complex arithmetic processor (CAP) or a DSP in accordance with a preferred embodiment of the invention;

In FIG. **2**, them is shown a schematic of a dual multiplier accumulator (MAC) suitable for use in the arithmetic engine shown in FIG. **1**;

In FIG. **3**, there is shown a first pair of adders suitable for the adder array shown in FIG. **1**;

In FIG. **4**, there is shown a second pair of adders suitable for the adder array shown in FIG. **1**; and

In FIG. **5**, there is shown a third pair of adders suitable for the adder array shown in FIG. **1**.

DETAILED DESCRIPTION OF THE DRAWINGS

Generally, the present invention provides an arithmetic engine apparatus optimized for complex arithmetic processing. While the arithmetic engine described is well suited for DSP applications, the use of such an arithmetic engine is not limited to DSP or complex arithmetic processing. The arithmetic invention may be accomplished by having two dual MACs in parallel, each connected to an adder array. Each

dual MAC can perform two multiplies and two addition operations simultaneously and can also sort its input data. The adder array can perform six addition operations simultaneously and is also capable of performing sorting and storing of peak values of input data.

The present invention can be more fully understood with reference to the figures. FIG. 1 illustrates a arithmetic engine **11** within a complex arithmetic processor (CAP) or digital signal processor (DSP) **10**. The arithmetic engine **11** includes a dual multiplier accumulator (dual MAC) **12**, a dual MAC **13**, and an adder array **14**. The adder array **14** is coupled to the parallel combination of dual MACs **12** and **13**. First dual MAC first input **16** and second dual MAC first input **17** are input lines that provide data to the dual MACs from the input data streams. First dual MAC second input **22** and second dual MAC second input **23** are input lines that can provide data to the respective dual MACs either from the input data streams, or from the adder array via first adder pair output data **74**. First adder pair output data **74** is a sub-set of the output of the adder array **20**. First dual MAC outputs **21** and second dual MAC outputs **19** are outputs of dual MAC **12** and dual MAC **13**, respectively. These dual MAC outputs are inputs to the adder array **14** and are elements of the output data streams. The adder array **14** can also receive data directly from the input data streams.

FIG. **2** illustrates a schematic of a dual MAC suitable for use in the arithmetic engine in FIG. **1**. The dual MAC in FIG. **2** describes specifically the dual MAC **12** in FIG. **1** by showing the inputs as first dual MAC first input **16**, first dual MAC second input **22**, and the output as first dual MAC outputs **21**. FIG. **2**. could represent dual MAC **13** in FIG. **1** if first dual MAC first input **16** and first dual MAC second input **22** were replaced by second dual MAC first input **17** and second dual MAC second input **23**, respectively. The output consisting of first dual MAC outputs **21** would, in such a case, be replaced by second dual MAC outputs **19**.

The dual MAC in FIG. **2** includes a dual multiplier **30** and a dual accumulator **34**. The dual multiplier **30** comprises a multiplier cross point switch **42**, multiplier registers **46**, register selector **50**, first parallel multiplier **52** and second parallel multiplier **54**. The cross point switch **42** allows any combination of inputs to be loaded into the multiplier registers **46** simultaneously. These registers are coupled to the register selector **50** which directs the appropriate multiplier register values **48** to both the first parallel multiplier **52**, and second parallel multiplier **54**. The outputs of the multipliers are coupled to both the MAC output selector **66** and the adder input registers **57** of the dual accumulator **34**.

Dual accumulator **34** comprises accumulator registers **56**, adder input registers **57**, an accumulator crosspoint switch **62**, first MAC adder **59**, second MAC adder **61**, condition code **63**, and MAC output selector **66**. The accumulator registers **56** are used to store the sum of previous MAC adder outputs which are first accumulator output **58** and second accumulator output **60** in FIG. **2**. The adder input registers **57** are used to store the outputs of the first and second parallel multiplier outputs **36** and **38** of the dual multiplier **30**. Both the accumulator registers **56** and the adder input registers **57** are coupled to the accumulator crosspoint switch **62**. The adder input registers **57** are also coupled to the MAC output selector **66**. The accumulator crosspoint switch **62** directs combinations of its inputs to both the first MAC adder **59** and the second MAC adder **61**. If the dual accumulator **34** is in accumulate mode, the outputs of the first and second MAC adders **59** and **61** are feedback to the accumulator registers **56** to store their outputs. The first and second MAC adders **59** and **61** are also coupled to the MAC output selector **66**. The second MAC adder **61**, which can also be used as a subtractor, is also coupled to the condition code **63**. The condition code **63** determines which of the second MAC adder **61** inputs is greater and uses this information to control the MAC output selector **66** which is coupled to its output. The MAC output selector **66** receives data from the first and second parallel multiplier outputs **36** and **38**, the adder input register values **51**, and from the first and second accumulator output **58** and **60**. The current mode of the dual MAC determines which of these outputs to send out on the first dual MAC outputs **21** in FIG. **2**. If the dual MAC is in the sort mode, the condition code output **64** is used to determine the appropriate order for the output of the adder input register values **51** on the first dual MAC outputs **21**.

To explain the workings of the FIG. **2** representation in further detail, data can enter the multiplier cross point switch **42** from the input data streams or from the first adder pair output data **74**. The input data streams consist of data pairs that come from the complex arithmetic processor memories. The first adder pair output data **74** comes from the first rank of adders inside the adder array **14**. The input data streams and the first adder pair output data enter the multiplier cross point switch **42** via the first dual MAC first input **16**, and the first dual MAC second input **22**, respectively. The multiplier cross point switch **42** routes the input data to the multiplier registers **46** as selected sources **44**. The cross point switch **42** is capable of moving multiple input values to multiple multiplier registers simultaneously. The register selector **50** receives the multiplier register values **48** from the multiplier registers **46** and sends the appropriate values to the first and second parallel multipliers **52** and **54** as selected multiplier operands **53**. The first and second parallel multipliers **52** and **54** multiply their respective inputs and output their results on first parallel multiplier output **36** and second parallel multiplier output **38**. If the dual MAC is in bypass mode, these outputs are sent directly to the MAC output selector **66** where they are sent out of the dual MAC via first dual MAC outputs **21**. If the dual MAC is in arithmetic mode, the first multiplier output **36** and the second multiplier output **38** are sent to the dual accumulator **34** and stored in the adder input registers **57**.

After data is stored in the input registers **57** in FIG. **2**, several things can happen. If the dual MAC is in arithmetic mode, the accumulator crosspoint switch **62** takes the two adder input registers values **51** and sends them both to the first MAC adder **59** and the second MAC adder **61** via the selected adder inputs **65**. In such a case, the second MAC adder **61** is being used as a subtractor, resulting in the sum and difference of the two values. The sum and difference appear on the first accumulator output **58** and the second accumulator output **60**, respectively, and enter the MAC output selector **66**, before exiting the dual MAC via the first dual MAC outputs **21**.

When the dual MAC in FIG. **2** is in the sort mode, the sum and difference of the adder input registers **57** are calculated as described above. The condition code **63** inputs the result from the second MAC adder **61** via the condition code input **67** (still being used as a subtractor), and determines which of the two adder input register values **51** was greater. A decision is sent to the MAC output selector **66** via the condition code output **64**. The decision is then used by the MAC output selector to determine how to arrange the adder input register values **51** on the first dual MAC output **21**. The largest and smallest values will always appear in the same location on the first dual MAC outputs **21**.

When the dual MAC in FIG. **2** is in the accumulator mode, the accumulator crosspoint switch **62** uses the adder

input register values **51** and the accumulator register values **50** as inputs. A top pair is sent to the first MAC adder **59** and a bottom pair is sent to the second MAC adder **61**. In the accumulate mode, the second MAC adder **61** is used as an adder and not a subtractor. The respective outputs of each of the first and the second MAC adders **59** and **61** are feedback to the accumulator registers **56** and added to the next values that are stored in the adder input registers **57**. Final accumulator values are output from the first and the second MAC adders **59** and **61**, sent to the MAC output selector **66**, and eventually out of the dual MAC via the first dual MAC outputs **21**.

FIG. **3** illustrates a first pair of adders suitable for the adder array **14** shown in FIG. **1**. The first rank of adders in FIG. **3** comprises a pair of cross point switches **84**, register files **86**, register selectors **88**, adders **92** and condition code determiners **94**. The cross point switches **84** accept input data streams to the arithmetic engine, outputs from the first dual MAC **21**, and outputs from the second dual MAC **19**. In addition to accepting external data, the cross point switches accept arithmetic engine output data. Outputs from the first, the second, and the third adder pairs, **74**, **174**, and **274**, respectively, are feedback to the first adder cross point switches **84**. The selected values **85** from the cross point switch **84** are stored in the first adder registers **86**. There must be at least eight registers in front of each adder **92** to efficiently perform a radix 8 butterfly algorithm. These registers buffer output data until it can be used by the adder. The need for buffering occurs because the input sequence of data does not match the sequence in which the data is consumed.

First adder operand A **90** and first operand B **91** in FIG. **3** are selected by the first adder register selector **88** from first adder registers **86** for use by the first adders **92**. Adders **92** can perform add or subtract operation. The first adder selectors **88** are also responsible for performing a sort operation as follows. First adder operands **90** and **91** are compared by first adder **92**. The sign of the result indicates which operand was larger. The sign of the result is determined by the condition code determiner **94** which feeds back the sort register select signal **97** to the first adder register selector **88**. This signal is used to select which operand **90** or **91** is output as the first operand sort **87**. In addition, the signal is used to select which value is output as first register sort **89** from a register pair which are not compared. These connections allow a sort chain to be established which can, in the preferred embodiment described, locate the five highest values (peak search) in one pass of the data through the arithmetic engine. The value of the peaks are on first operands sort **87** while the locations of the peaks are on first registers sort **89**.

FIG. **4** illustrates a second pair of adders suitable for the adder array **14** shown in FIG. **1**. The second pair of adders in FIG. **4** comprises a pair of cross point switches **184**, register files **186**, register selectors **188**, adders **192** and condition code determiners **194**. The cross point switches **184** accept input data streams to the arithmetic engine, outputs from the first dual MAC **21**, and outputs from the second dual MAC **19**. In addition to accepting external data, the cross point switches accept arithmetic engine output data. Outputs from the first, the second, and the third adder pairs, **74**, **174**, and **274**, respectively, are feedback to the second adder cross point switches **184**. The selected values **185** from the cross point switch **184** are stored in the second adder registers **186**. Second adder operand A **190** and second operand B **191** are selected by the second adder register selector **188** from second adder registers **186** for use by the second adders **192**. Adders **192** can perform add or subtract operation. The second adder selectors **188** are also responsible for performing the sort operation. Second adder operands **190** and **191** are compared by second adder **192**. The sign of the result indicates which operand was larger. The sign of the result is determined by the condition code determiner **194** which feeds back the sort register select signal **197** to the second adder register selector **188**. This signal is used to select which operand **190** or **191** is output as the second operand sort **187**. In addition, the signal is used to select which value is output as second register sort **189** from a register pair which are not compared. These connections allow a sort chain to be established which can, in the preferred embodiment described, locate the five highest values (peak search) in one pass of the data through the arithmetic engine. The value of the peaks are on second operands sort **187** while the locations of the peaks are on second registers sort **189**.

FIG. **5** illustrates a third pair of adders suitable for the adder array **14** shown in FIG. **1**. The third pair of adders in FIG. **5** comprises a pair of cross point switches **284**, register files **286**, register selectors **288**, adders **292** and condition code determiners **294**. The cross point switches **284** accept input data streams to the arithmetic engine, outputs from the first dual MAC **21**, and outputs from the second dual MAC **19**. In addition to accepting external data, the cross point switches accept arithmetic engine output data. Outputs from the first, the second, and the third adder pairs, **74**, **174**, and **274**, respectively, are feedback to the first adder cross point switches **284**. The selected values **285** from the cross point switch **284** are stored in the third adder registers **286**. Third adder operand A **290** and third operand B **291** are selected by the third adder register selector **288** from third adder registers **286** for use by the third adders **292**. Adders **292** can perform add or subtract operation. The third adder selectors **288** are also responsible for performing the sort operation. Third adder operands **290** and **291** are compared by third adder **292**. The sign of the result indicates which operand was larger. The sign of the result is determined by the condition code determiner **294** which feeds back the sort register select signal **297** to the third adder register selector **288**. This signal is used to select which operand **290** or **291** is output as the third operand sort **287**. In addition, the signal is used to select which value is output as third register sort **289** from a register pair which are not compared. These connections allow a sort chain to be established which can, in the preferred embodiment described, locate the five highest values (peak search) in one pass of the data through the arithmetic engine. The value of the peaks are on third operands sort **287** while the locations of the peaks are on third registers sort **289**.

As an example illustrating how the dual MACs **12** and **13** in FIG. **1** work with the adder array **14**, a description of the radix 8 butterfly will be given. The radix 8 butterfly function acts on eight complex numbers producing eight complex outputs. Data passes through the FIG. **2** dual multipliers **30**, dual accumulator **34**, and each rank of the adder arrays **95**, **195**, **295** in FIG. **3** using all of the resources of the arithmetic engine each instruction. During each instruction, a complex multiply is performed in dual MACs **12** and **13** and six add operations occur in the adder array **14** (add or subtract). Twelve instructions are required to fill the arithmetic engine **11** registers with data and produce the first output. After arithmetic engine pipeline is full, output data is produced every instruction, as can be seen in the detailed description below.

This detailed discussion of how data is sequenced through the arithmetic engine during a radix 8 butterfly refers to

Table 1, Radix 8 Pipeline Sequence. Table 1 shows how data is transferred from register to register for each instruction. Data enters the arithmetic engine at the dual MAC cross point switch **42** in FIG. **2** and is stored in the multiplier input registers **46**. There are four registers assigned to each multiplier. Registers **M11–M14** are assigned to first parallel multiplier **52**. Registers **M21–M24** are assigned to second parallel multiplier **54** of the first dual MAC **12**. Registers **M31–M34** and **M41–M44** are assigned to the multipliers of the second dual MAC **13**. Input data consists of complex data samples **X7**, **Y7** and complex coefficients **Zx7**, **Zy7**. Instruction 1 loads eight registers simultaneously through the cross point switch (**M11**, **M41**, **M21**, **M31**, **M13**, **M33**, **M23**, **M43**).

Instruction 2 loads the next complex data pair (**X6,Y6**), (**Zx3,Zy3**) into the multiplier registers **46** and multiplies data from the previous fetch and stores it into the adder input registers **57**, which are designated **A11**, **A12** for the first dual Mac **12** and **A21**, **A22** for the second dual MAC **13**.

Instruction 3 fetches the third complex pair (**X5,Y5**) (**Zx5,Zy5**) and completes the complex multiply of the first data sample by adding and subtracting the previous adder input registers **57** (**A11**, **A12**, **A21**, **A22**). The results are stored in the first adder registers **86** of the adder array **14** (**A38,A48**) (in FIG. **3**).

TABLE 1

	RADIX 8, 8 CYCLE BUTTERFLY PIPEFILL SEQUENCE				
Input to MAC	Output of Multiplier	Output of MAC	Output of 1st Adder Pair	Output of 2nd Adder Pair	Output of 3rd Adder Pair
Instruction #: 1					
X7→M11,M41 Y7→M21,M31 Zx7→M13,M33 Zy7→M23,M43					
Instruction #: 2					
X6→M11,M41 Y6→M21,M31 Zx3→M13,M33 Zy3→M23,M43	M11*M13→A11 M21*M23→A12 M31*M33→A21 M41*M43→A22				
Instruction #: 3					
X5→M11,M41 Y5→M21,M31 Zx5→M13,M33 Zy5→M23,M43	M11*M13→A11 M21*M23→A12 M31*M33→A21 M41*M43→A22	A11+A12→A38 A21–A22→A48			
Instruction #: 4					
X4→M11,M41 Y4→M21,M31 Zx1→M13,M33 Zy1→M23,M43	M11*M13→A11 M21*M23→A12 M31*M33→A21 M41*M43→A22	A11+A12→A34 A21–A22→A44			
Instruction #: 5					
X3→M11,M41 Y3→M21,M31 Zx6→MI3,M33 Zy6→M23,M43	M11*M13→A11 M21*M23→A12 M31*M33→A21 M41*M43→A22	A11+A12→A37 A21–A22→A47	A34–A38→M42 A44–A48→M22		
Instruction #: 6					
X2→M11,M41 Y2→M21,M31 Zx2→M13,M33 Zy2→M23,M43	M11*M13→A11 M21*M23→A12 M31*M33→A21 M41*M43→A22	A11+A12→A33 A21–A22→A43	A34+A38→A57 A44+A48→A67		
Instruction #: 7					
X1→M11,M41 Y1→M21,M31 Zx4→M13,M33 Zy4→M23,M43	M11*M13→A11 M21*M23→A12 M31*M33→A21 M41*M43→A22	A11+A12→A36 A21–A22→A46	A33–A37→M12 A43–A47→M32		
Instruction #: 8					
X0→A31 Y0→A41	M11*M13→A11 M21*M23→A12 M31*M33→A21 M41*M43→A22	A11+A12→A32 A21–A22→A42	A33+A37→A53 A43+A47→A63		
Instruction #: 9					
X15→M11,M41 Y15→M21,M31 Zx7→M13,M33 Zy7→M23,M43	M14*M12→A11 M24*M22→A12 M34*M32→A21 M44*M42→A22	A11+A12→A35 A21–A22→A45	A32+A36→A55 A42+A46→A65	A53+A57→A75 A63+A67→A85	

TABLE 1-continued

RADIX 8, 8 CYCLE BUTTERFLY PIPEFILL SEQUENCE

Input to MAC	Output of Multiplier	Output of MAC	Output of 1st Adder Pair	Output of 2nd Adder Pair	Output of 3rd Adder Pair
Instruction #: 10					
X14→M11,M41	M11*M13→A11	A11+A12→A54	A31+A35→A51	A53–A57→A87	
Y14→M21,M31	M21*M23→A12	A11–A12→A64	A41+A45→A61	A63–A67→A77	
Zx3→M13,M33	M31*M33→A21	A21+A22→A68			
Zy3→M23,M43	M41*M43→A22	A21–A22→A58			
Instruction #: 11					
X13→M11,M41	M11*M13→A11	A11+A12→A38	A32–A36→A66	A51+A55→A71	
Y13→M21,M31	M21*M23→A12	A21–A22→A48	A42–A46→A56	A61+A65→A81	
Zx5→M13,M33	M31*M33→A21				
Zy5→M23,M43	M41*M43→A22				
Instruction #: 12					
X12→M11,M41	M11*M13→A11	A11+A12→A34	A31–A35→A52	A51–A55→A73	
Y12→M21,M31	M21*M23→A12	A21–A22→A44	A41–A45→A62	A61–A65→A83	
Zx1→M13,M33	M31*M33→A21				
Zy1→M23,M43	M41*M43→A22				
Instruction #: 13					
X11→M11,M41	M11*M13→A11	A11+A12→A37	A34–A38→M42	A54–A58→A86	A71+A75→X0
Y11→M21,M31	M21*M23→A12	A21–A22→A47	A44–A48→M22	A64–A68→A78	A81+A85→Y0
Zy6→M13,M33	M31*M33→A21				
Zy6→M23,M43	M41*M43→A22				
Instruction #: 14					
X10→M11,M41	M11*M13→A11	A11+AI2→A33	A34+A38→A57	A54+A58→A76	A71–A75→X4
Y10→M21,M31	M21*M23→A12	A21–A22→A43	A44+A48→A67	A64+A68→A88	A81–A85→Y4
Zx2→M13,M33	M31*M33→A21				
Zy2→M23,M43	M41*M43→A22				
Instruction #: 15					
X9→M11,M41	M11*M13→A11	A11+A12→A36	A33–A37→M12	A52+A56→A72	A73+A77→X2
Y9→M21,M31	M21*M23→A12	A21–A22→A46	A43–A47→M32	A62+A66→A84	A83–A87→Y2
Zx4→M13,M33	M31*M33→A21				
Zy4→M23,M43	M41*M43→A22				
Instruction #: 16					
X8→A31	M11*M13→A11	A11+A12→A32	A33+A37→A53	A52–A56→A74	A73–A77→X6
Y8→A41	M21*M23→A12	A21–A22→A42	A43+A47→A63	A62–A66→A82	A83+A87→Y6
	M31*M33→A21				
	M41*M43→A22				
Instruction #: 17					
X23→M11,M41	M14*M12→A11	A11+A12→A35	A32+A36→A55	A53+A57→A75	A72+A76→X1
Y23→M21,M31	M24*M22→A12	A21–A22→A45	A42+A46→A65	A63+A67→A85	A82–A86→Y1
Zx7→M13,M33	M34*M32→A21				
Zy7→M23,M43	M44*M42→A22				
Instruction #: 18					
X22→M11,M41	M14*M12→A11	A11+A12→A54	A31+A35→A51	A53–A57→A87	A72–A76→X5
Y22→M21,M31	M24*M22→A12	A11–A12→A64	A41+A45→A61	A63–A67→A77	A82+A86→Y5
Zx3→M13,M33	M34*M32→A21	A21+A22→A68			
Zy3→M23,M43	M44*M42→A22	A21–A22→A58			
Instruction #: 19					
X21→M11,M41	M14*M12→A11	A11+A12→A38	A32–A36→A66	A51 +A55→A71	A74–A78→X3
Y21→M21,M31	M24*M22→A12	A21–A22→A48	A42–A46→A56	A61 +A65→A81	A84–A88→Y3
Zx5→M13,M33	M34*M32→A21				
Zy5→M23,M43	M44*M42→A22				
Instruction #: 20					
X20→M11,M41	M14*M12→A11	A11+A12→A34	A31–A35→A52	A51–A55→A73	A74+A78→X7
Y20→M21,M31	M24*M22→A12	A21–A22→A44	A41–A45→A62	A61–A65→A83	A84+A88→Y7
Zx1→M13,M33	M34*M32→A21				
Zy1→M23,M43	M44*M42→A22				
Instruction #: 21					
X19→M11,M41	M14*M12→A11	A11+A12→A37	A34–A38→M42	A54–A58→A86	A71+A75→X8
Y19→M21,M31	M24*M22→A12	A21–A22→A47	A44–A48→M22	A64–A68→A78	A81+A85→Y8
Zx6→M13,M33	M34*M32→A21				
Zy6→M23,M43	M44*M42→A22				

TABLE 1-continued

RADIX 8, 8 CYCLE BUTTERFLY PIPEFILL SEQUENCE

Input to MAC	Output of Multiplier	Output of MAC	Output of 1st Adder Pair	Output of 2nd Adder Pair	Output of 3rd Adder Pair
Instruction #: 22					
X18→M11,M41	M14*M12→A11	A11+A12→A33	A34+A38→A57	A54+A58→A76	A71–A75→X12
Y18→M21,M31	M24*M22→A12	A21–A22→A43	A44+A48→A67	A64+A68→A88	A81–A85→Y12
Zx2→M13,M33	M34*M32→A21				
Zy2→M23,M43	M44*M42→A22				
Instruction #: 23					
X17→M11,M41	M14*M12→A11	A11+A12→A36	A33–A37→M12	A52+A56→A72	A73+A77→X10
Y17→M21,M31	M24*M22→A12	A21–A22→A46	A43–A47→M32	A62+A66→A84	A83–A87→Y10
Zx4→M13,M33	M34*M32→A21				
Zy4→M23,M43	M44*M42→A22				

Instruction 4 fetches the fourth complex pair (X4,Y4) (Zx1,Zy1), completes the complex multiply of the second sample by adding and subtracting the previous adder input registers **57**, and stores the results in the first adder registers **86** of the adder array **14** (A34, A44).

Instruction 5 fetches the fifth complex pair (X3,Y3) (Zx6,Zy6), completes the complex multiply of the third sample, and stores the results in the first adder registers **86** (A37, A47). The previous values loaded into the first adder registers in instructions 3 and 4 (A34, A38, A44, A48) are subtracted and stored back in the dual MAC multiplier registers **46** (M42, M22) to be used later in instruction 9. This instruction demonstrates the feedback properties of the arithmetic engine. The feedback is applied to maintain complete utilization of the arithmetic engine resources while calculating the radix 8 butterfly.

Instruction 6 fetches the sixth complex pair (X2,Y2) (Zx2,Zy2), completes the complex multiply of the fourth sample, and stores the results in the first adder registers **86** (A33, A43). The values loaded into the first adder registers in instructions 3 and 4 (A34, A38, A44, A48) are added and stored in the second adder registers **186** of the second rank adder block (FIG. 4 A57 and A67).

Instruction 7 fetches the seventh complex pair (X1,Y1) (Zx4,Zy4), completes the complex multiply of the fifth sample, and stores the results in the first adder registers **86** (A36, A46). The previous values loaded into the first adder registers in instructions 5 and 6 (A33, A37, A43, A47) are subtracted and stored back in the dual MAC multiplier registers **46** (M12, M32) to be used later in instruction 9. This instruction again demonstrates the feedback properties of the arithmetic engine.

Instruction 8 fetches the pair (X0,Y0) and stores them directly into A31 and A41 of the first adder registers **86**. The complex multiply of the sixth sample is completed, and the values loaded into the first adder registers in instruction 5 and 6 (A33, A37, A43, A47) are added and loaded into the second adder registers **186** (A53, A63).

Instruction 9 fetches the ninth complex pair (X15,Y15) (Zx7,Zy7), completes the complex multiply of the seventh instruction, adds the previous values of the first adder registers in instruction 7 and 8 (A32, A36, A42, A46) and loads them into the second adder registers **186** (A55, A65). The previous values loaded in the second adder registers in instruction 6 and 8 (A53, A57, A63, A67) are added and stored in the third adder registers **286** (FIG. 5 A75 and A85). This same process continues through instruction 12.

After instruction 12, the pipeline is full and the final results of the radix eight butterfly are available beginning with instruction 13. Instruction 13 operates like instruction 9 through 12 with the exception of a final operation. Instruction 13 takes the previous values loaded into the third adder registers **286** in instructions 9 and 11 (A71, A75, A81, A85), adds them together, and sends the result (X0, Y0) to memory. This result is the first output value of the radix 8 butterfly. Instruction 14 outputs the second result (X4, Y4) and the process continues for a single butterfly through instruction 20 where the last value is output (X7, Y7).

In summary, the Fast Fourier Transform is a fundamental signal processing application. The speed at which this transform can be computed often determines which digital signal processor is chosen for a given application. Higher radix butterflies decrease the execution time of the FFT by increasing throughput, provided there are sufficient resources to store intermediary values. The arithmetic engine contemplated here, while optimized for a radix 8 butterfly, can also perform radix 2 and radix 4 butterflies efficiently.

The significance of this arithmetic engine described lies in its architecture. The architecture is designed to input and operate on complex data in ordered pairs. Each adder pair is designed to operate on a complex number. There are three ranks of adder pairs because of the Radix 8 butterfly requirement. The output of the first adder pair **74** provides radix 2 results, the output of the second adder pair **174** provides radix 4 results, and the output of the third adder pair **274** provides radix 8 results.

Another important capability designed into the hardware is high speed data sorting. Most arithmetic engines simply pass data through from the input to the output. The engine described here differs because it recirculates data within the engine by feeding the outputs back to the inputs. This feature allows full use of computing resources and is essential for parallel sorting operations where all of the adders and switches are active. Use of a cross point switch allows any combination of inputs to be loaded simultaneously into any combination of registers. Typically, simple multiplexers are used which only allow one or two registers to be loaded at a time.

Thus, there has also been provided, in accordance with an embodiment of the invention, an arithmetic engine that fully satisfies the aims and advantages set forth above. While the invention has been described in conjunction with a specific embodiment, many alternatives, modifications, and variations will be apparent to those of ordinary skill in the art in light of the foregoing description. Accordingly, the invention is intended to embrace all such alternatives, modifications,

and variations as fall within the spirit and broad scope of the appended claims.

What is claimed is:

1. An arithmetic engine comprising:

a first dual multiplier accumulator (MAC) for receiving input data at a first dual MAC first input and for producing first dual MAC output data at a first dual MAC output, wherein the first dual MAC comprises:

a dual multiplier for receiving a plurality of sources and for producing first and second parallel multiplier outputs, the dual multiplier including a multiplier cross point switch for receiving the plurality of sources and for producing selected sources at a plurality of multiplier cross point switch outputs, wherein the dual multiplier comprises:

a plurality of multiplier registers coupled to the multiplier cross point switch outputs, the plurality of multiplier registers for storing the selected sources as multiplier register values;

a register selector coupled to the plurality of multiplier registers, the register selector for choosing selected multiplier operands from the multiplier register values; and

first and second parallel multipliers coupled to the register selector, the first and the second parallel multipliers for receiving pairs of selected multiplier operands and multiplying the pairs of selected multiplier operands to produce the first and the second parallel multiplier outputs, respectively;

an accumulator coupled to the dual multiplier, the accumulator for receiving the first and second parallel multiplier outputs, selecting particular first and second parallel multiplier outputs in a condition code determiner, and for producing the first dual MAC output data therefrom, the accumulator comprising:

a plurality of adder registers for storing the first and the second parallel multiplier outputs and a plurality of accumulator registers for storing a first and a second accumulator output;

an accumulator cross point switch coupled to the plurality of adder registers and the plurality of accumulator registers, the accumulator cross point switch for producing selected accumulator register and adder register values from the first and the second parallel multiplier outputs and the first and the second accumulator outputs;

first and second parallel MAC adders coupled to the accumulator cross point switch, the first and the second parallel MAC adders for receiving pairs of selected accumulator register and adder register values and adding the pairs of selected accumulator register and adder register values to produce the first and the second accumulator adder outputs; and

a MAC output selector coupled to the first and to the second parallel MAC adders and to the first and to the second parallel multipliers, the MAC output selector for receiving the first and the second accumulator outputs and the first and the second parallel multiplier outputs and for producing the first dual MAC output data; and

a first dual MAC second input;

a second dual MAC coupled in parallel to the first dual MAC, the second dual MAC for receiving the input data at a second dual MAC first input and for producing second dual MAC output data at a second dual MAC output, wherein the second dual MAC comprises a second dual MAC second input; and

an adder array coupled to both the first dual MAC and to the second dual MAC, the adder array for receiving the input data, the first dual MAC output data, and the second dual MAC output data and for producing arithmetic engine output data at an adder array output, wherein the first dual MAC second input is coupled to the adder array output, the first dual MAC output is coupled to the adder array output, the second dual MAC second input is coupled to the adder array output, and the second dual MAC output is coupled to the adder array output.

2. An arithmetic engine as claimed in claim 1, wherein the second parallel MAC adder produces a condition code input to the condition code determiner, and the condition code determiner produces a condition code output to the MAC output selector.

3. An arithmetic engine as claimed in claim 1, wherein the adder array comprises:

a first adder pair for receiving the input data, the first dual MAC output data, the second dual MAC output data and first, second, and third adder pair outputs, and for producing first adder pair outputs;

a second adder pair coupled to the first adder pair, the second adder pair for receiving the input data, the first dual MAC output data, the second dual MAC output data, and the first, the second, and the third adder pair outputs and for producing second adder pair outputs; and

a third adder pair coupled to the second adder pair, the third adder pair for receiving the input data, the first dual MAC output data, the second dual MAC output data, and the first, the second, and the third adder pair outputs and for producing the arithmetic engine output data at the adder array output.

4. An arithmetic engine as claimed in claim 3, wherein each adder of the first adder pair comprises:

a first adder cross point switch for producing first selected values of the input data, the first MAC output data, the second MAC output data and the first, the second, and the third adder pair outputs;

first adder registers coupled to the first adder cross point switch, the first registers for storing the first selected values;

a first adder register selector coupled to the first adder registers, the first adder register selector for receiving and choosing first operands and for producing a first adder sum output from the first selected values;

a first adder coupled to the first adder register selector, the first adder for receiving and adding the first operands and for producing one of the first adder pair outputs; and

a first condition code determiner for receiving a first adder condition code input and for providing a first condition code to the first adder register selector.

5. An arithmetic engine as claimed in claim 3, wherein each adder of the second adder pair comprises:

a second adder cross point switch for producing second selected values of the input data, the first dual MAC output data, the second dual MAC output data, and the first, the second, and the third adder pair outputs;

second adder registers coupled to the second adder cross point switch, the second adder registers for storing the second selected values;

a second adder register selector coupled to the second adder registers, the second adder register/selector for receiving and choosing second operands from the second selected values;

a second adder coupled to the second adder register selector, the second adder for receiving and adding the second operands and for producing one of the second adder pair outputs; and

a second condition code determiner for receiving a second adder condition code input and for providing a second condition code to the second adder register selector.

6. An arithmetic engine as claimed in claim 3, wherein each adder of the third adder pair comprises:

a third adder cross point switch for producing third selected values of the input data, the first dual MAC output data, the second dual MAC output data, and the first, the second and the third adder pair outputs;

third adder registers coupled to the third adder cross point switch, the third adder registers for storing the third selected values;

a third adder register selector coupled to the third adder registers, the third adder register selector for receiving and choosing third operands from the third selected values;

a third adder coupled to the third adder register selector, the third adder for receiving and adding the third operands and for producing the arithmetic engine output data; and

a third condition code determiner for receiving a third condition code input and for providing a third condition code to the third adder register selector.

7. A complex arithmetic processor including an arithmetic engine comprising:

a first dual multiplier accumulator (MAC) for receiving input data at a first dual MAC first input and for producing first dual MAC output data at a first dual MAC output, wherein the first dual MAC comprises:

a dual multiplier for receiving a plurality of sources and for producing first and second parallel multiplier outputs, the dual multiplier including a multiplier cross point switch for receiving the plurality of sources and for producing selected sources at a plurality of multiplier cross point switch outputs, wherein the dual multiplier comprises:

a plurality of multiplier registers coupled to the multiplier cross point switch outputs, the plurality of multiplier registers for storing the selected sources as multiplier register values;

a register selector coupled to the plurality of multiplier registers, the register selector for choosing selected multiplier operands from the multiplier register values; and

first and second parallel multipliers coupled to the register selector, the first and the second parallel multipliers for receiving pairs of selected multiplier operands and multiplying the pairs of selected multiplier operands to produce the first and the second parallel multiplier outputs, respectively;

an accumulator coupled to the dual multiplier, the accumulator for receiving the first and second parallel multiplier outputs, selecting particular first and second parallel multiplier outputs in a condition code determiner, and for producing the first dual MAC output data therefrom, wherein the accumulator comprises:

a plurality of adder registers for storing the first and the second parallel multiplier outputs and a plurality of accumulator registers for storing a first and a second accumulator output;

an accumulator cross point switch coupled to the plurality of adder registers and the plurality of accumulator registers, the accumulator cross point switch for producing selected accumulator register and adder register values from the first and the second parallel multiplier outputs and the first and the second accumulator outputs;

first and second parallel MAC adders coupled to the accumulator cross point switch, the first and the second parallel MAC adders for receiving pairs of selected accumulator register and adder register values and adding the pairs of selected accumulator register and adder register values to produce the first and the second accumulator adder outputs; and

a MAC output selector coupled to the first and to the second parallel MAC adders and to the first and to the second parallel multipliers, the MAC output selector for receiving the first and the second accumulator outputs and the first and the second parallel multiplier outputs and for producing the first dual MAC output data;

a first dual MAC second input;

a second dual MAC coupled in parallel to the first dual MAC, the second dual MAC for receiving the input data at a second dual MAC first input and for producing second dual MAC output data at a second dual MAC output, wherein the second dual MAC comprises a second dual MAC second input; and

an adder array coupled to both the first dual MAC and to the second dual MAC, the adder array for receiving the input data, the first dual MAC output data, and the second dual MAC output data and for producing arithmetic engine output data at an adder array output, wherein the first dual MAC second input is coupled to the adder array output, the first dual MAC output is coupled to the adder array output, the second dual MAC second input is coupled to the adder array output, and the second dual MAC output is coupled to the adder array output.

8. A complex arithmetic processor as claimed in claim 7, wherein the second parallel MAC adder produces a condition code input to the condition code determiner, and the condition code determiner produces a condition code output to the MAC output selector.

9. A complex arithmetic processor as claimed in claim 7, wherein the adder array comprises:

a first adder pair for receiving the input data, the first dual MAC output data, the second dual MAC output data and first, second, and third adder pair outputs, and for producing first adder pair outputs;

a second adder pair coupled to the first adder pair, the second adder pair for receiving the input data, the first dual MAC output data, the second dual MAC output data, and the first, the second, and the third adder pair outputs and for producing second adder pair outputs; and

a third adder pair coupled to the second adder pair, the third adder pair for receiving the input data, the first dual MAC output data, the second dual MAC output data, and the first, the second, and the third adder pair outputs and for producing the arithmetic engine output data at the adder array output.

10. A complex arithmetic processor as claimed in claim 9, wherein each adder of the first adder pair comprises:

a first adder cross point switch for producing first selected values of the input data, the first MAC output data, the second MAC output data and the first, the second, and the third adder pair outputs;

first adder registers coupled to the first adder cross point switch, the first registers for storing the first selected values;

a first adder register selector coupled to the first adder registers, the first adder register selector for receiving and choosing first operands and for producing a first adder sum output from the first selected values;

a first adder coupled to the first adder register selector, the first adder for receiving and adding the first operands and for producing one of the first adder pair outputs; and

a first condition code determiner for receiving a first adder condition code input and for providing a first condition code to the first adder register selector.

11. A complex arithmetic processor as claimed in claim 9, wherein each adder of the second adder pair comprises:

a second adder cross point switch for producing second selected values of the input data, the first dual MAC output data, the second dual MAC output data, and the first, the second, and the third adder pair outputs;

second adder registers coupled to the second adder cross point switch, the second adder registers for storing the second selected values;

a second adder register selector coupled to the second adder registers, the second adder register selector for receiving and choosing second operands from the second selected values;

a second adder coupled to the second adder register selector, the second adder for receiving and adding the second operands and for producing one of the second adder pair outputs; and

a second condition code determiner for receiving a second adder condition code input and for providing a second condition code to the second adder register selector.

12. A complex arithmetic processor as claimed in claim 9, wherein each adder of the third adder pair comprises:

a third adder cross point switch for producing third selected values of the input data, the first dual MAC output data, the second dual MAC output data, and the first, the second and the third adder pair outputs;

third adder registers coupled to the third adder cross point switch, the third adder registers for storing the third selected values;

a third adder register selector coupled to the third adder registers, the third adder register selector for receiving and choosing third operands from the third selected values;

a third adder coupled to the third adder register selector, the third adder for receiving and adding the third operands and for producing the arithmetic engine output data; and

a third condition code determiner for receiving a third condition code input and for providing a third condition code to the third adder register selector.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 5,522,085
DATED : May 28, 1996
INVENTOR(S) : Calvin Wayne Harrison et al.

It is certified that error appears in the above-indentified patent and that said Letters Patent is hereby corrected as shown below:

In column 15, claim 5, line 2, delete "register/selector" and substitute --register selector--.

Signed and Sealed this

Twenty-fourth Day of September, 1996

Attest:

BRUCE LEHMAN

Attesting Officer

Commissioner of Patents and Trademarks

US005422836A

United States Patent [19]

Beichter et al.

[11] **Patent Number: 5,422,836**

[45] **Date of Patent: Jun. 6, 1995**

[54] **CIRCUIT ARRANGEMENT FOR CALCULATING MATRIX OPERATIONS IN SIGNAL PROCESSING**

[75] Inventors: **Jörg Beichter; Ulrich Ramacher**, both of München, Germany

[73] Assignee: **Siemens Aktiengesellschaft**, Munich, Germany

[21] Appl. No.: **50,103**

[22] PCT Filed: **Nov. 4, 1991**

[86] PCT No.: **PCT/DE91/00858**

§ 371 Date: **May 7, 1993**

§ 102(e) Date: **May 7, 1993**

[87] PCT Pub. No.: **WO92/09040**

PCT Pub. Date: **May 29, 1992**

[30] **Foreign Application Priority Data**

Nov. 15, 1990 [DE] Germany 40 36 455.0

[51] **Int. Cl.**[6] **G06F 15/347**

[52] **U.S. Cl.** **364/736**; 364/754

[58] **Field of Search** 364/736, 736.5, 754

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,611,305	7/1986	Iwase	364/736
4,815,019	3/1989	Bosshart	364/736.5
4,949,292	8/1990	Hoshino et al.	364/736
4,958,312	9/1990	Ang et al.	364/754
5,175,702	12/1992	Beraud et al.	364/736
5,179,531	1/1993	Yamaki	364/736

FOREIGN PATENT DOCUMENTS

3735654	6/1988	Germany .
2226899	7/1990	United Kingdom .

OTHER PUBLICATIONS

"Multiplizierer und Akkumulator U 1520 PC 001", by B. Schönfelder et al, vol. 37, No. 10, (1988), Berlin, pp. 626–630.

"A Systolic/Cellular Computer Architecture For Linear Algebraic Operations", by J. G. Nash, Proceedings of the 1985 IEEE Int. Conf. on Robotics & Automation, IEEE Computer Society Press, New York, Mar. 25, 1985, St. Louis Mo., pp. 779–784.

"Programmable Architectures For Matrix & Signal Processing", by Brad Hamilton, Proceedings of IEEE Region 5 Conf. Spanning the Peaks of Electrotechnology, IEEE Computer Society Press, New York, Mar. 21, 1988, pp. 116–126.

"The Matrix Transform Chip", by S. K. Rao, Proceedings of 1989 IEEE Int. Conf. On Computer Design: VLSI In Computers & Processors; IEEE Computer Society Press, New York, Oct. 2, 1989, Cambridge, Mass., pp. 86–89.

"Vector Reduction Methods For Arithmetic Pipelines", by L. M. Ni et al, Proceedings of the 6th Symposium on Computer Arithmetic, IEEE Computer Society Press, New York, Jun. 20, 1983, pp. 144–150.

"Design of A 1st Generation Neurocomputer", edited by U. Ramacher et al, VLSI Design of Neural Networks, Kluwer Academic Publishers, Nov. 1990, 40 pages.

Primary Examiner—David H. Malzann

Attorney, Agent, or Firm—Hill, Steadman & Simpson

[57] **ABSTRACT**

Circuit arrangement for calculating matrix operations, such as those which recur frequently in signal processing, specifically in conjunction with neural networks, having a systolic array of multipliers and adders, downstream from which a recursive accumulator is connected. In addition to products, sums and differences of matrices, this circuit arrangement also allows squares, absolute magnitudes of sums and differences and squares of sums and differences of two matrices to be calculated very efficiently. Furthermore, with the aid of the recursive accumulator, it is possible to transpose matrices, to calculate row sums and column sums, and to search for minimum or maximum matrix elements.

6 Claims, 4 Drawing Sheets

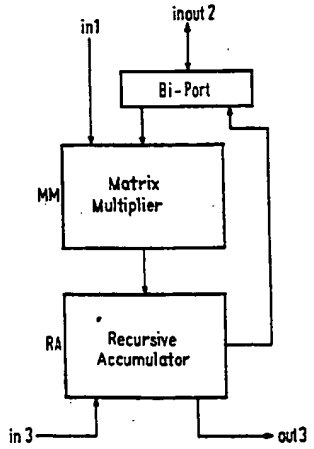


FIG 1

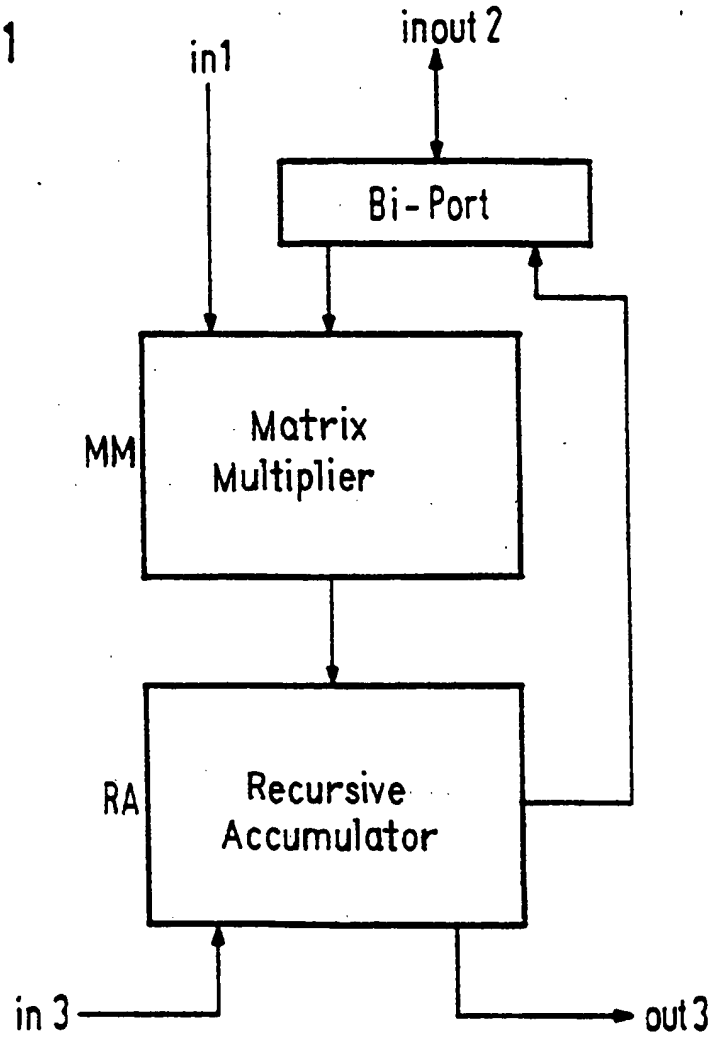

in1
inout 2
Bi-Port
MM
Matrix
Multiplier
RA
Recursive
Accumulator
in 3
out 3

FIG 2

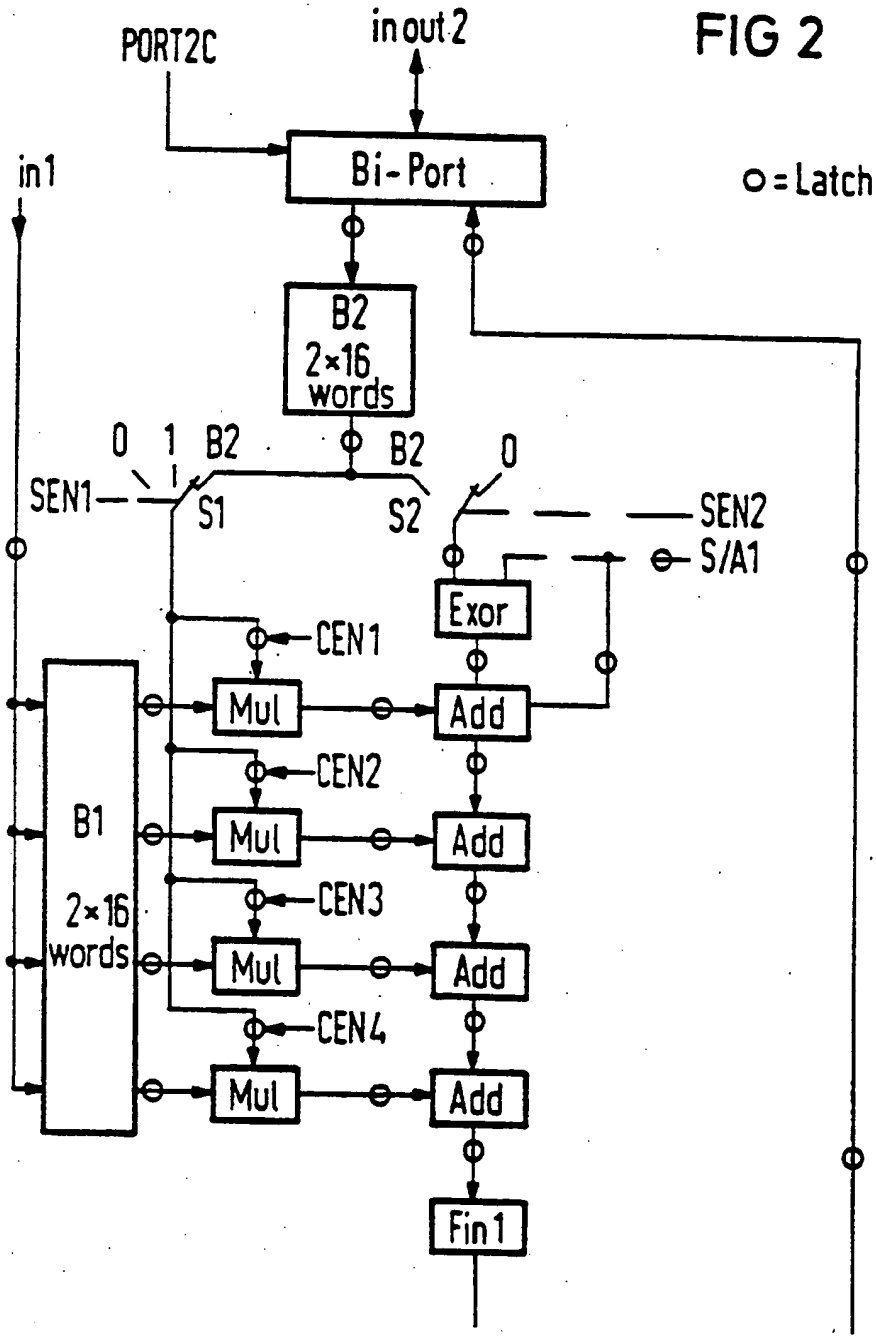

PORT2C
in out 2
in1
Bi-Port
o = Latch
B2
2×16
words
0
1
B2
B2
0
SEN1
S1
S2
SEN2
S/A1
Exor
CEN1
Mul
Add
CEN2
B1
2×16
words
CEN3
CEN4
Fin 1

FIG 3

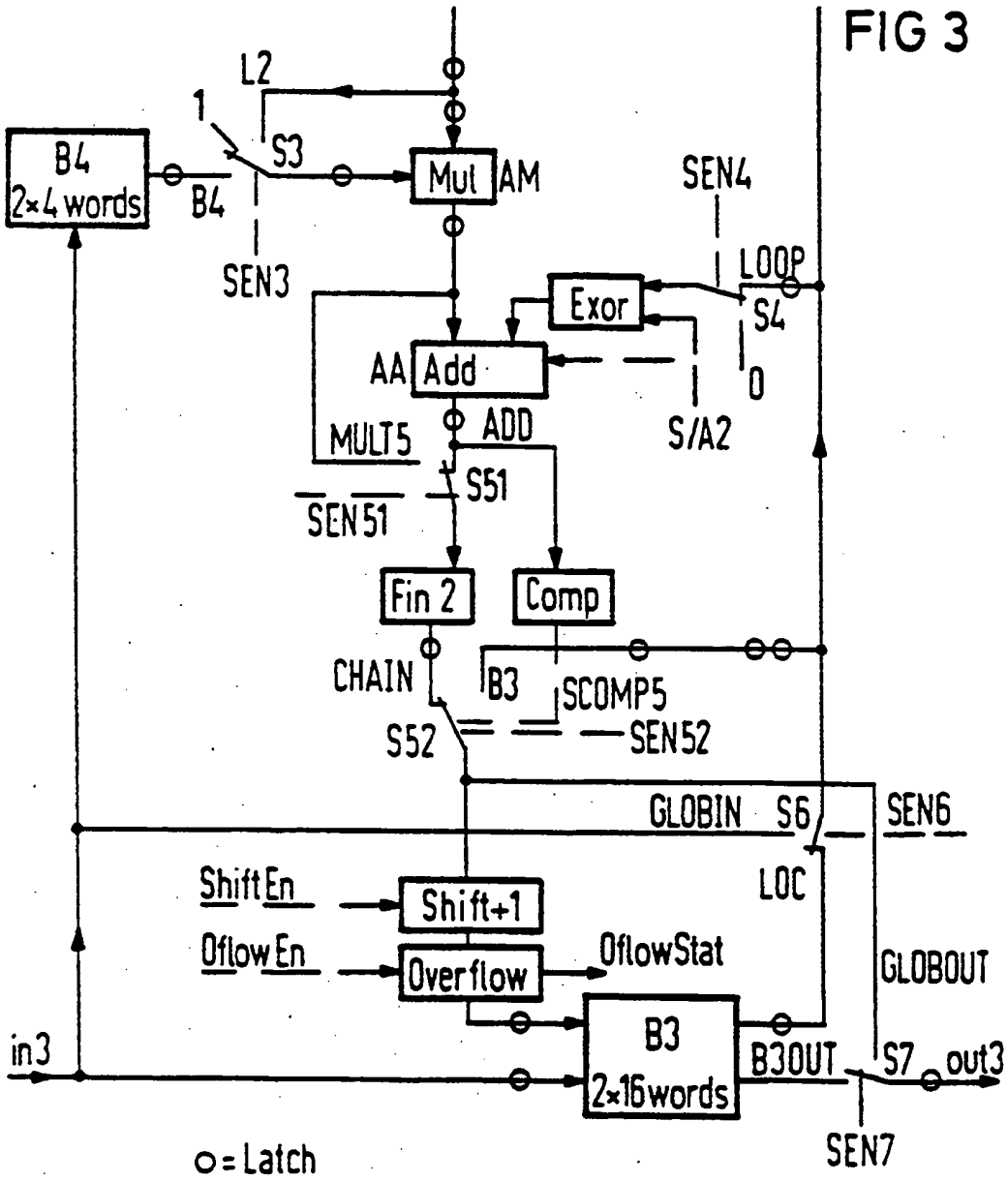

L2
1
S3
B4
2×4 words
B4
Mul
AM
SEN4
SEN3
LOOP
Exor
S4
AA
Add
0
MULT5
ADD
S/A2
S51
SEN51
Fin 2
Comp
CHAIN
B3
SCOMP5
S52
SEN52
GLOBIN
S6
SEN6
LOC
ShiftEn
Shift+1
OflowEn
Overflow
OflowStat
GLOBOUT
in3
B3
2×16words
B3OUT
S7
out3
SEN7
o=Latch

FIG 4

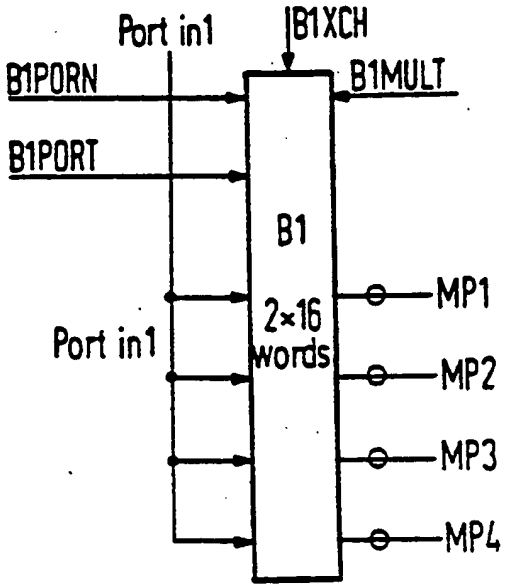


FIG 5

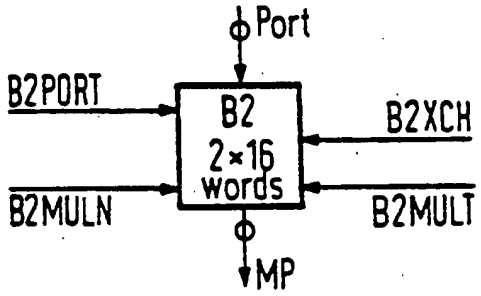


FIG 6

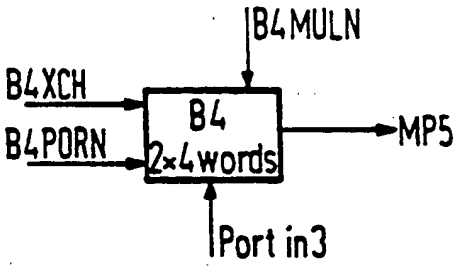
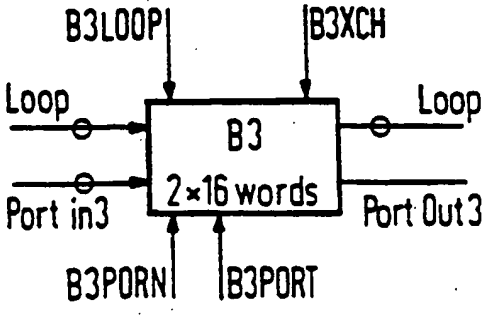


FIG 7



CIRCUIT ARRANGEMENT FOR CALCULATING MATRIX OPERATIONS IN SIGNAL PROCESSING

BACKGROUND OF THE INVENTION

The invention relates to a circuit arrangement for calculating matrix operations which recur frequently, such as those in signal processing, especially in the context of neural networks. Since the computation operations which are required for calculating neural networks can be reduced to a comprehensible number of elementary matrix operations, it is sensible from the point of view of the necessary high computation speed in the execution of these operations to implement such computation operations in hardware, rather than to carry them out with the aid of software.

A prior art which is closest to the invention is reproduced in the publication by U. Ramacher, "Design of a first Generation Neurocomputer", VLSI Design of Neural Networks, edited by U. Ramacher, U. Rückert, Kluwer Academic Publishers, Nov. 1990. This publication describes a circuit arrangement which is constructed from a systolic arrangement of multipliers and adders. This systolic arrangement makes it possible to calculate matrix products, the matrices which are to be multiplied being split into blocks of size 4×4, and it being possible to multiply submatrices of this size with the aid of the systolic arrangement in each case. The computation operations which can be carried out using this circuit arrangement are suitable for the calculation of specific neural network types, for example multilayer perceptron networks with feedback.

The disadvantages of this circuit arrangement, as it is described in the publication by U. Ramacher 1990, are primarily that:

the transposition, addition and subtraction are not supported by matrices,

result matrices cannot be squared or cannot be multiplied by a scalar, and that

the calculation of row and column sums and the search for extreme matrix elements are not supported by this circuit arrangement.

Furthermore, in the case of this circuit arrangement, there is no monitoring of the value range of the matrix coefficients, and the value ranges of the matrix elements are not limited if an overflow occurs.

SUMMARY OF THE INVENTION

The object on which the invention is based is to specify a circuit arrangement by means of which the described disadvantages of the prior art are overcome, and which supports the calculation of matrix products and the multiplication of matrix products by scalars, as well as the squaring of matrix products, sums and different formation of matrices, the multiplication of matrix sums and matrix differences by scalars, forming the magnitude of matrix sums and matrix differences, squaring matrix sums and matrix differences, transposition of matrices and matrix products, the calculation of row and column sums of matrices, and the search for extreme, that is to say minimum and maximum, matrix elements. This object is achieved by means of a circuit arrangement for calculating matrix operations, having a matrix multiplier and a recursive accumulator, which is connected downstream from this matrix multiplier, and having the following features:

the matrices which are to be processed by the circuit arrangement are partitioned into k x k submatrices;

the overall circuit has first and second inputs, one output and one input/output;

the matrix multiplier has the following units:

dual-port first and second memory units for the storage and transposition of input submatrices, the first memory unit being connected to the first input and the second memory unit being connected to the input/output, and the first memory unit having k independent memories;

a systolic arrangement, which is connected to the first and second memory units via inputs which are controlled by control signals, which arrangement has k multipliers and k downstream-connected adders for the multiplication and addition/subtraction of input submatrices, the first memory unit storing the rows of the submatrix in k separate sub-memory units, and supplying this data via k separate supply lines to the multipliers, so that each multiplier, with the downstream-connected adder successively, and in a synchronized manner, carries out a multiplication of a matrix element from the corresponding row of the first memory unit with the matrix element of a row of the submatrix from the second memory element, and the results are accumulated, via the adder chain which has the k adders of the matrix multiplier, to form product sums;

a first final adder of the carry-select type is in this case provided at the end of this adder chain, and the recursive accumulator has the following units:

third and fourth memory units, both of which are connected to the second input, the third memory unit storing the weightings and the fourth memory unit storing the coefficients;

a further multiplier for the multiplication of the product sums in the first final adder by the coefficients or for squaring the product sums, which further multiplier is connected at a second input either to the output of the first final adder or to the output of the fourth memory unit;

a further adder, which is connected downstream from the further multiplier, for the component-based addition of product sums with the weightings from the third memory;

a second final adder, which is connected downstream from the further adder and is of the carry-select type, and

a recursive loop via the further adder, the second final adder or a comparator, a shifter and an overflow-monitoring circuit, as well as via the third memory unit for component-based MIN/MAX comparison of the result submatrices with a submatrix which is present at the third input, and for overflow regulation of the weightings which are stored in the third memory unit.

A systolic arrangement of multipliers and adders is likewise provided in the circuit arrangement according to the invention. In contrast to the circuit arrangement known from the prior art, in the circuit arrangement according to the invention, a recurslye accumulator is connected downstream from this systolic arrangement of multipliers and adders. It is possible to carry out a considerably more comprehensive class of computation operations with the aid of this recursive accumulator. In particular, using the circuit arrangement according to the invention, it is possible to multiply matrix products by scalars, to square matrices or matrix products, to form sums or differences of matrices and to multiply by scalars, to calculate the absolute magnitude of matrix

sums and matrix differences and their squares, and to transpose matrices. Furthermore, the circuit arrangement according to the invention allows the calculation of row and column sums of matrices and matrix products and sums or differences of matrices. Finally, this circuit arrangement makes it possible to search for maximum and minimum matrix elements in previously calculated matrices. The present invention is also a circuit arrangement for calculating transpositions, row sums and column sums of matrices and for searching for extreme matrix elements, which has a multiplier and an adder which is connected downstream from this multiplier and links the output of the multiplier to an output of this circuit arrangement, in a recursive manner.

The present invention is further a circuit arrangement for calculating matrix operations, which has a matrix multiplier and a recursive accumulator, which is connected downstream from the matrix multiplier, for calculating transpositions, row sums and column sums and for searching for extreme matrix elements. The accumulator has a multiplier and an adder, which is connected downstream from this multiplier and links the output of the multiplier to an output of this circuit arrangement, in a recursive manner.

Advantageous developments of the present invention are as follows. The matrix multiplier has two dual-port memory units and a systolic chain of multipliers and adders, this chain being connected downstream from these memory units.

The adders of the systolic chain are of the carry-save type.

The matrix multiplier has a systolic chain of multipliers and adders, at whose end a first final adder of the carry-select type is provided.

The recursive accumulator has a second final adder which is connected downstream from the adder and is of the carry-select type.

The recursive accumulator has a comparator which compares the result of the adder with zero.

BRIEF DESCRIPTION OF THE DRAWINGS

The features of the present invention which are believed to be novel, are set forth with particularity in the appended claims. The invention, together with further objects and advantages, may best be understood by reference to the following description taken in conjunction with the accompanying drawings, in the several Figures of which like reference numerals identify like elements, and in which:

FIG. 1 shows a signal-flow diagram of a circuit arrangement for calculating matrix operations.

FIG. 2 shows a preferred implementation of a sub-circuit of the circuit from FIG. 1, which is designated, in FIG. 1, as a matrix multiplier.

FIG. 3 shows a preferred implementation of a sub-circuit of the circuit arrangement from FIG. 1, which is designated, in FIG. 1, as a recursive accumulator.

DESCRIPTION OF THE PREFERRED EMBODIMENT

The FIGS. 4, 5, 6 and 7 show signal-flow diagrams for driving memory units B1, B2, B3 and B4.

As FIG. 1 shows, the circuit arrangement according to the invention comprises two sub-circuits which are designated as a matrix multiplier MM and as a recursive accumulator KA respectively. The overall circuit has two inputs, which are designated by in1 and in3 respectively, an output which is designated by out3, and a bidirectional interface which is designated by inout2. The bidirectional interface can be monitored with the aid of the signal Port2c (see FIG. 2). The coefficients of two matrices which are to be multiplied with one another are supplied via the interfaces in1 and inout2 respectively and the memories B1 and B2 respectively, connected thereto, to a systolic arrangement comprising four multipliers and four adders. The input data of the multipliers of the systolic arrangement are in this case selected with the aid of the switch S1, which is controlled via the signal SEN1. The adder chain of the systolic arrangement is connected to the output of an EXOR gate whose first input is connected to the memory B2 via the switch S2, which is controlled with the aid of the signal SEN2. The other inputs of the multipliers are connected to the memory B1. This memory is a combination of four independent memories which in each case comprise two-times four words. Latches are provided in all cases for buffer storage of data and computation results in the systolic arrangement and in the overall circuit. The latches do not carry out any computational functions but ensure merely the synchronous sequencing of the overall circuit, and are required in order to implement algorithmically dependent time delays. Located at the end of the adder chain of the systolic arrangement is a final adder Fin1, which combines the carry digits of the carry-save adders, which are designated by Add, into the carry-select format. A preferred embodiment of the multipliers within the systolic arrangement is described in the work by U. Ramacher, 1990.

A recursive accumulator, which is shown in FIG. 3, is connected in the overall circuit to the matrix multiplier which is shown in FIG. 1 and is described in more detail in FIG. 2 (see FIG. 1). A multiplier AM, downstream from which an adder AA is connected, is located at the input of this recursive accumulator. One input of this multiplier is connected to the output of the final adder of the matrix multiplier. The second input of the multiplier can be connected to the output of the memory B4 or to the output of the final adder, or can have the constant value 1 depending on the position of the switch S3, which is controlled via the signal SEN3. The second input of the adder AA which is connected downstream from this multiplier is connected to the output of a second EXOR gate. One input of this second EXOR gate is connected to the signal LOOP via the switch S4 which is controlled via the signal SEN4. This signal LOOP is identical to the signal at the input in3 or to the output of the memory B3, depending on the position of the switch S6, which is controlled via the signal SEN6. The output of the memory B3 is also designated as LOC in FIG. 3. The output of the adder AA is connected to a second final adder Fin2 via the switch S51 which is controlled via the signal SEN 51. Connected in parallel with this final adder Fin2 is a comparator Comp which compares the output of the adder AAwith 0 and hence acts as a mathematical-sign decision element. This mathematical-sign signal, together with the control signal SEN52, determines the position of the switch S52 which, depending on the switch position, connects the output signal, which is designated by CHAIN, of the final adder Fin2 or the loop signal, which is designated by B3 and is delayed via three latches, to a shifter which is controlled by the signal ShiftEn. The output of the shifter is connected to an overflow control circuit Overflow, whose output is connected to the memory unit B3.

The signal path via the adder AA, the final adder Fin2 or the comparator Comp, the shifter and the overflow control circuit, as well as via the memory unit B3 and the EXOR gate whose output forms the second input of the adder AA, represents a recursive loop by means of which the computation operations which are novel in comparison with the prior art can be carried out.

In contrast to the circuit arrangement which is described in the work by U. Ramacher, 1990, the circuit arrangement according to the invention now also makes possible matrix addition and component-based min/max comparison, in addition to matrix-matrix multiplication. The input matrices, split into 4×4 submatrices, are locally transposed and are then added or multiplied. The components of the result matrix can optionally be squared or multiplied by a scalar. Either a global accumulation or a min/max comparison in the row or column direction can be applied to the 4×4 submatrices produced in this way.

The value range of the 16-bit weighting values are monitored for an overflow and, if an overflow occurs, the value is automatically limited. Using a special shift device, the value range of the weightings can be objectively changed, for example halved or doubled in steps, in order thus to avoid an overflow and to utilize the 16-bit word width of the weighting memory more effectively.

The development of the circuit arrangement according to the invention for calculating matrix operations is based on the idea of distributing the overall computation work between a predetermined number of elementary circuit arrangements, and of partitioning the large matrices which are to be multiplied or added into 4×4 submatrices. Each elementary circuit arrangement thus processes only a specific row and column region (submatrices) of the matrix which is held in local memories. The calculation of large matrices is combined from the processing of the 4×4 submatrices. During this combination, the indices of the submatrices are expanded to the overall effective ranges of the large matrices. Operations which achieve the solution for large matrices from the calculation of submatrices are local and global accumulation and local or global min/max comparison of the components.

The circuit arrangement according to the invention carries out the following simple computation operations which, linked in different ways, implement the neural algorithms. In this case, the term submatrix is used to designate 4×4 matrix blocks which are produced by partitioning large matrices.

1. Multiplication of input submatrices

1.1 Submatrix A (in the memory B1) multiplied by submatrix B in the memory (B2);

1.2 Submatrix A (in the memory B1) multiplied by the unit matrix;

2. Addition/subtraction of input submatrices

2.1 Submatrix B (in the memory B2) is added to the submatrix A (in the memory B1), previously multiplied by the unit matrix;

2.2 Submatrix B (in the memory B2) is subtracted from submatrix A (in the memory B1), previously multiplied by the unit matrix;

2.3 Submatrix B (in the memory B2) pass [sic] through the adder chain, no addition;

3. Multiplication of the result submatrices from 1.1 to 2.2.

3.1 Multiplication of the result submatrix by a coefficient (a dedicated coefficient for each submatrix column).

3.2 No multiplication (multiplication by 1).

3.3 Multiplication of the individual submatrix components by themselves (squaring).

4. Addition/accumulation of result submatrices from 3.1 to 3.3.

4.1 Addition of the result submatrix to the submatrix which is stored in the memory unit B3 (local accumulation).

4.2 Addition of the result submatrix to the submatrix which is applied to the input in3 at the same time (distributed accumulation).

5. Component-based min/max comparison for the result submatrix from 3.1 to 3.3.

5.1 Min/max comparison between the results from 3.1 to 3.3 and the submatrix stored in the memory unit B3.

5.2 Min/max comparison between the submatrix from 3.1 to 3.3 and a submatrix which is applied to the input in3 at the same time.

6. Overflow regulation of the weightings which are stored in the memory unit B3.

6.1 Overflow recognition and value saturation of the weightings which are to be limited to 16 bits.

6.2 Controlled, global reformatting of all weighting submatrices.

The computation operations 1.1 to 6.2 which are implemented in the circuit arrangement are carried out in different parts of the overall circuit and can be connected to one another in specific combinations in order to support different neural algorithms.

The following can be combined:

in each case one operation 1.1 to 2.2 with

in each case one operation 3.1 to 3.3 and

in each case one operation 4.1 to 5.2.

The two operations 6.1 and 6.2 can be combined with 4.1.

The execution of the individual computation operations with the aid of the circuit arrangement, and the control of the circuit arrangement are described in detail in the following text.

Computation operation 1.1: submatrix in the memory B1 multiplied by the submatrix in the memory B2.

The two matrices A and B are loaded into the memories B1 and B2. Switch S1 is in the position B2, switch S2 is in the position 0 and the signal S/A1 has the value 0. The memory B2 supplies the data B (1,1), B (2,1), ..., B (4,4), that is to say the matrix elements of the left, top 4×4 submatrix of the matrix B, in a distributed manner with each clock cycle over 16 clock cycles. The control signals CEN1, ..., CEN4 at the input registers of the multipliers have the values 0 and are thus not active except at the following times: in clock cycle 1, CEN1 is active and causes the loading of the input register of the uppermost multiplier with the matrix element B (1,1). In the next clock cycle, CEN2 is active and loads the input of the second multiplier with the matrix element B (2,1). In the third and fourth clock cycles, the input registers of the other two multipliers are loaded with CEN3 and CEN4. In the fifth clock cycle, CEN1 is active again and loads the matrix element B (1,2) in the first multiplier. This distribution process continues cyclically and ends after the 16th clock cycle when the matrix element B (4,4) is loaded into the fourth multiplier by means of CEN4=active.

In parallel with this process, the memory unit B1 holds the submatrix A (1,1), ..., A (4,4) in a column-based manner in four separate sub-memory units and supplies these data via four separate supply lines to the multipliers of the systolic arrangement (see FIG. 2). The first multiplier receives its first matrix element A (1,1) from the memory B1 at the same time as the matrix element B (1,1). In the three subsequent clock cycles, the memory B1 supplies the matrix elements A (2,1), A (3,1) and A (4,1) via the same line. Between the fifth and eighth clock cycles, when the matrix element B (1,2) is also loaded at the multiplier, the application to the matrix elements A (1,1), ..., A (4,1) is repeated. After 16 clock cycles, that is to say once the sequence of the matrix elements A (1,1), . .., A (4,1) has been applied to the multipliers of the systolic arrangement four times, the reading for the first multiplier of the chain is completed. The second multiplier receives the matrix elements A (1,2) ..., A (4,2) from the memory B1. The third and the fourth multipliers respectively receive the matrix elements A (1,3), . .., A (4,3) and A (1,4), ..., A (4,4) respectively from the memory B1 with in each case one further clock cycle delay. Overall, a delay of three clock cycles results between the first and the fourth multipliers of the chain. In consequence, the last multiplier receives the value of the matrix element A (4,4) for the fourth and last time in the 19th clock cycle.

Each multiplier can start one multiplication of two matrix elements of the matrices A or B respectively per clock cycle. Such a multiplication lasts for seven clock cycles. The accumulation of the results is accomplished via the adder chain of the matrix multiplier in FIG. 2. In clock cycle 8, the uppermost, first adder receives the product A (1,1) . B (1,1) of the first multiplier, adds the zero from switch S2 thereto and passes the result to the subsequent second adder, in the following clock cycle (clock cycle 9). At the same time as it receives the sum of the first adder from the second multiplier, said second adder receives the product B (1,2) . A (2,1) which it adds to the sum (clock cycle 10). The products B (1,3) . A (3,1) and B (1,4) . A (4,1) are also added in the following two clock cycles. In clock cycle 12, the sum of the four products of the matrix elements of the first row of the submatrix A and the first column of the submatrix B are present at the output of the fourth and final adder. The sums which are still missing are added in the further clock cycles. The overall result matrix is completely calculated when the sum of the products of the matrix elements of the fourth row of the matrix A and the matrix elements of the fourth column of the matrix B appears at the output of the adder chain, in the 28th clock cycle.

Computation operation 1.2: submatrix in the memory B1 multiplied by the unit matrix (no multiplication).

The computation operation 1.2 runs in an analogous manner to the computation operation 1.1, the second matrix B being replaced by the unit matrix. The switch S1 is switched cyclically between the value 1 and the value 0 for this purpose. S1 is at 1 in clock cycles 1, 6, 11 and 16, and is at 0 in all the other clock cycles.

Computation operation 2.1: submatrix in the memory B2 added to the submatrix in the memory B1.

In order to carry out the computation operation 2.1, the computation operation 1.2, which causes a multiplication of the matrix A which is present in the memory B1, by the unit matrix, is expanded by the addition of a matrix B which is present in the memory B2. In this case, the switch S2 is in the position B2. The control signal S/A1 has the value zero. All the other steps correspond to those in the computation operation 1.2.

Computation operation 2.2: Submatrix in the memory B2 subtracted from the submatrix in the memory B1.

In this case, the sequence is completely analogous to the sequence during the computation operation 2.1. Only a change in the mathematical sign is required, which is carried out by the signal S/A1 being given the value 1. In consequence, the EXOR gate together with the adder connected downstream from it forms the two's-complement of the data which are present in the memory B2.

Computation operation 2.3: Loading the data which are present in the memory B2 into the recurs ire accumulator without addition, that is to say circumventing the systolic adder chain.

The computation operation 2.3 is carried out in an analogous manner to the computation operation 2.1, a zero matrix being supplied, however, instead of a unit matrix. This is done by the switch S1 being in the position zero during all the clock cycles.

Computation operation 3.1: Multiplication of a result submatrix by a coefficient.

In the case of this computation operation, the systolic adder chain supplies the sums, which are mentioned in the description of the computation operation 1.1, of the products from the matrix elements of the lines of matrix A by the columns of matrix B, to be precise in the sequence such that the sum of the products from the matrix elements of the first line of matrix A and the matrix elements of the first column of the matrix B is calculated first, and the sum of the products from the matrix elements of the fourth row of the matrix A and the fourth column of the matrix B is calculated last, starting from the 27th clock cycle. The upper 19 bits of the data are present in the carry-save form and must be converted into the binary form (final adder Fin1) before they can be supplied as input data to the first multiplier AM of the recursive accumulator. At the same time, a delay of one clock cycle is produced by the final adder Fin1. Two further clock cycles are produced to provide the data at both inputs of the multiplier AM.

The sums of the products of matrix elements coming from the final adder Fin1 arrive at the multiplier AM in the sequence described above, and are multiplied by the coefficients K(1), K(2), K(3) and K(4), which are stored in the memory B4. In this case, the following products are formed in series:

$$X(1) \cdot P(1,1), K(1) \cdot P(2,1), K(1) \cdot P(3,1), K(1) \cdot P(4,1),$$
$$K(2) \cdot P(1,2), K(2) \cdot P(2,2), \ldots, K(4) \cdot P(4,4).$$

In this case, the product P(i,J) is equal to the sum of the products of the matrix elements in the i-th row of the matrix A and the matrix elements in the k-th column of the matrix B. The multiplier requires seven clock cycles, a new multiplication being started in each clock cycle. The results appear at the output of the multiplier AM from the 21st to the 37th clock cycles. In this case, the switch S3 is always in position B4. Computation operation 3.2: No multiplication.

The switch S3 is in the position 1, as a result of which the value 1 is allocated to all the coefficients K(i) for i =1, ..., 4. The remaining sequence is identical to the computation operation 3.1.

Computation operation 3.3: Squaring of the matrix components.

The switch S3 is in the position L2. The values P(i,j) coming from the adder chain are applied to both inputs of the multiplier AM. The multiplicand word and the multiplier word are thus identical. If all the other steps are carried out in an analogous manner to computation operation 3.1, then the product sums P(i,J) appear in the position of the coefficients K(i) and the product matrix is multiplied by itself. In accordance with the computation operation 1.2, the product matrix can, preferably also be one of the original matrices.

Computation operation 4.1.1: Local accumulation of the matrix from 3.1 to 3.3

The product sums P(1,1), P(2,1), ..., P(4,4) coming from the adder chain of the matrix multiplier are added in a component-based manner to the data S(1,1), S(2,1), ..., S(4,4) which are stored in the memory B3. The new values of S(i,J) which thus result are subsequently stored in the memory B3 again (fetch and add).

In this case, the switch S4 is in the position LOOP, the switch S51 is in the position ADD, the switch S52 is in the position CHAIN, and the switch S6 is in the position LOC. The control signal S/A2 has the value zero here, the control signals ShiftEn and OflowEn (see FIG. 3) both being set to zero initially, here. The data S(1,1), S(2,1), ..., S(4,4) are read in this sequence out of the part of the memory B3 which is connected to the recursive loop. Said data are passed via the switches S6 and S4 and the EXOR gate to the adder AA, which is constructed from a series of full-adders. In clock cycle 22, S(1,1) is added to P(1,1), and S(2,1) is added to P(2,1) in the following clock cycle.

The carry word and sum word are linked in the downstream-connected final adder Fin2 to form a single binary value. The 50-bit-wide final adder Fin2 is a complex circuit having a long delay time, as a result of which, if possible, it is implemented at only one point and not after every adder stage. The delay time in the final adder Fin2 is two clock-cycle periods.

The results, the new values S(1,1), ..., S(4,4), are passed via the switch S52 to the shifter. If the value of the signal ShiftEn is zero, then the data remain unchanged. If OflowEn=0, the downstream-connected overflow controller also has no effect on the data. Six clock cycles after the old values of S(i,J) have been read out of the memory B3, the new values of S(i,J) are written in.

Computation operation 4.1.2: Local accumulation: Resetting of the recursive loop.

A zero is added to the product sums P(i,J) in each case, instead of the old values S(i,J). In this case, the switch S4 is at zero. The remaining sequence is the same as in the computation operation 4.1.1.

Computation operation 4.2: Distributed accumulation of the matrix from 3.1 to 3.3.

The switch S4 is in the position LOOP, the switch S51 is in the position ADD, the switch S52 is in the position CHAIN, and the switch S6 is in the position GLOBIN. In contrast to the operation 4.1.1, the old values of S(1,1), ..., S(4,4) are not read out of the memory B3 but are taken from the input in3. The accumulation passes via the switches S6 and S4 and the adders AA and Fin2 as well as the switches S51 and S52. After the switch S52, the results, the new values S(1,1), ..., S(4,4), are not written via the shifter into the memory B3 but are passed directly to the output out3 via the switch S7. A delay time of five clock cycles is required for this purpose.

Computation operation 5.1: Component-based min/-max comparison of the result submatrix from 3.1 to 3.3 with the submatrix which is stored in the part of the memory B3 which belongs to the recursive loop.

The switch S4 is in the position LOOP, and the switch S6 is in the position LOC. The signal S/A2 has the value 1 and the EXOR gate and the downstream-connected adder stage AA are used to form the two's complement of the data passing via the switch S4. The switch S51 is in the position MULT5 and the position of the switch S52 is produced from the result of the comparator Comp, which tests whether the result of the adder AA is greater than or equal to zero. The adder determines the difference between the value originating from the multiplier AM and the value of the memory B3. If this difference is positive, the comparator positions the switch S52 into the position MULT5 during MAX comparison, and into the position ADD during MIN comparison. If, in contrast, the calculated difference is negative, then the value from the memory B3 is greater than the product at the output of the multiplier AM, and the switch positions are interchanged. The result which is selected via the switch S52 is stored in the memory B3 without any further change.

Computation operation 5.2: Component-based MIN/MAX comparison of the result submatrix from 3.1 to 3.3 with the submatrix which is applied to the input in3 at the same time.

In contrast to the computation operation 5.1, the switch S6 is here in the position GLOBIN and the switch S7 is in the position GLOBOUT. All the other steps are analogous to the computation operation 5.1.

Computation operation 6.1: Overflow recognition and value saturation for the 16-bits of data stored in the memory B3 (weightings).

The word length of the weightings in neural networks is usually limited to 16 bits. Overflow recognition and saturation are used to suppress the limits which can be represented being exceeded or undershot during the learning process. The data coming from the switch S52 are tested in the overflow unit for overflow and, possibly, are saturated, if OflowEn=1 is set by the controller (16 bit wordlength). If OflowEn=0, no saturation is carried out and the data path has a 50-bit length. An overflow counter, which can be read by the controller, is incremented whenever an overflow is recognized (OflowStat =1).

Computation operation 6.2: Reformatting the weighting matrices.

Before data are written into the part of the memory B3 which belongs to the recursive loop, they can be shifted by one position to the right or left. The numerical range which can be represented can thus be matched dynamically during the learning process. The drive via the signal ShiftEn takes place from the controller.

The matrix operations which can be carried out with the aid of the circuit arrangement have thus been described. The following text is intended to describe the operations in the memory units B1, ..., B4 comprehensively. These further operations are carried out in the memory units B1, ..., B4 in order to permit simple and effective data transportation between the circuit arrangement and its environment (for example, simultaneous loading, computation and unloading). Furthermore, the matrix components are sorted and, if desired, transposed, for processing in the circuit arrangement. The individual memory units are selected in the manner shown in the signal-flow diagrams in FIGS. 4 to 7.

7. Operations in the matrix memory B1
7.1.1 Load, not transposed
7.1.2 Load transposed
7.2.1 Change in the memory halves
7.2.2 No change in the memory halves
7.3 Read the data
8. Operations in the matrix memory B2
8.1 Load the data
8.2.1 Change in the memory halves
8.2.2 No change in the memory halves
8.3.1 Read, not transposed
8.3.2 Read, transposed
9. Operations in the matrix memory B3
9.1 Fetch and add, in the half of the memory B3 belonging to the loop
9.2.1 Change in the memory halves
9.2.2 No change in the memory halves
9.3.1 Load/read in the memory half which belongs to the interface (Port), not transposed
9.3.2 Load/read in the memory half which belongs to the interface, transposed
10. Operations in the coefficient buffer B4
10.1 Load the data
10.2 Read the data
10.3 Change in the memory halves
11. Control of the bidirectional interface inout2.
11.1 Interface switched to input
11.2 Interface switched to output

In conjunction with most computation operations, the interface inout2 is used as an input (11.1), and only for short, precisely specified steps is it used as an output (11.2).

The operations in the memory units are described comprehensively in the following text. The terms port-side and loop-side are used for brevity in this description. Port-side is intended to mean a memory half which belongs to the interface, while loop-side means a memory half facing the recursive loop.

The data A(i,J) are supplied from the exterior via the interface in1 and pass into the memory B1. Their sequence is A(1,1), A(1,2), ..., A(4,4). The memory B1 holds these 16 matrix elements such that the matrix elements A(1,1), ..., A(4,1), that is to say the first column of the 4×4 submatrix of A, are in the top part of the memory and can subsequently be allocated to the multiplier 1. The matrix elements A(1,2), ... A(4,2), that is to say the elements of the second column, are stored in the second memory row for the multiplier 2, etc. The word-line signals are passed systolically from memory word to memory word, using a built-in sequencer, in order to select the memory cells, so that only one memory word is written per clock cycle. The sequencer is triggered by a trigger pulse B1PORN from the controller. Furthermore, B1PORT=0.

7.1.2 Load, transposed

In contrast to 7.1.1, the data are stored in B1 such that the data A(1,1), ..., A(1,4) are stored in the first memory row, and the data A(2,1), ... A(2,4), are stored in the second memory row, etc. A dedicated sequencer, which is triggered by B1PORT (B1PORN=0), is required for the changed sequence.

7.2.1 Change in the memory halves

The allocations of the two halves of the memory unit B1, the one half to the input in 1 (loading), the other half to the multiplier chain (reading), are interchanged. The controller initializes the change in the memory halves via the control bit B1XCH. The signal B1XCH is passed from memory row to memory row in the memory unit B1 with a delay of one clock cycle in order to compensate for the delay time in the adder chain. It is possible to change the allocation of the memory halves only before a submatrix operation, that is to say at the earliest every 16 clock cycles.

7.2.2 No change in the buffer halves

The submatrix stored in B1 is required a plurality of times successively for calculations (learning process). Overwriting of these data is prevented if the change in the memory halves is omitted. The data which are stored on the multiplier side are maintained until the next change in the memory halves. B1XCH does not change its value.

7.3 Read the data

Reading takes place via a sequencer which is triggered via B1MULN. See 1.1 to 2.3 for the sequence for this purpose.

8.1 Load B2

B2 has 16 memory cells. The data B(i,j) are stored in the memory in series. The memory fields are selected by means of a sequencer, B2PORN being used for triggering (see 7.1.1).

8.2.1 Change in the memory halves

The change in the memory halves is controlled by the controller via B2XCH. The change covers all the buffer halves at the same time. A change is possible with every new submatrix operation, at the earliest every 16 clock cycles.

8.2.1 No change in the buffer halves

If the value of B2XCH is maintained, the buffer halves are not changed.

8.3.1 Read, not transposed

The memory cells are read in the same sequence in the other memory half, in parallel with writing: Trigger pulse B2MULN.

8.3.2 Read, transposed

The sequence of the data read corresponds to the transposed matrix: Trigger pulse B2MULT.

9.1 Fetch-and-add in B3, loop-side

For the fetch-and-add operation, the data which are to be written in 9.1 are read from the same memory cells of the memory B3 in which they will subsequently be stored again, 6 clock cycles in advance (trigger: B3LOOP).

9.2.1 Change in the buffer halves

The change is controlled by the controller via B3XCH (see 8.2.1).

9.2.1 No change in the buffer halves

The value of B3XCH is maintained (see also 8.2.2).

9.3.1 Read, port-side, not transposed

The data located in B3 (port-side) are read in the sequence S (1,1), S (2,1), ..., S (4,4) and are replaced by values which are simultaneously present at the input. The sequencer is triggered by B3PORN.

9.3.2 Read, port-side, transposed

The data located in B3 (port-side) are read in the sequence S (1,1), S (1,2), ..., S (4,4) and are replaced by values which are present at the input of B3 at the same time. The sequencer is triggered by B3PORT.

10.1 Load the data in B4

The data in B4 (coefficients) are supplied to the circuit arrangement via the interface in3 (OUT3). Four clock cycles are required for loading. The trigger pulse for the sequencer is B4PORN.

10.2 Read the data

The trigger pulse for reading is B4MULN.

10.3 Change in the memory halves

Four further memory fields are provided for storage of a second set of coefficients. Selection takes place via B4XCH.

11.1 Interface INOUT2 switched to input.

The input drivers of the interface INOUT2 are always active.

11.2 Interface INOUT2 switched to output.

The changeover of the interface INOUT2 to output operation takes place via an external control bit PORT2C, in order to prevent external driver conflicts.

The circuit arrangement described relates to a circuit module which, connected together with identical circuit modules, produces a coprocessor for carrying out matrix operations. This coprocessor is supplied with the necessary control signals from a controller circuit.

The internal construction of the controller and also of the memory units is described in the diploma thesis by U. Hachmann, "Controller Architektur für einen Neuroemulator in Form eines systolischen Arrays" [Controller architecture for a neural emulator in the form of a systolic array], Department of Electrical Engineering Components, Dortmund University, Dortmund 1990.

Specialist terms which are in normal use and are used herein for designating digital circuits are explained, for example, in Kai Hwang, "Computer Arithmetics Principles, Architecture and Design", John Wiley & Sons, 1979.

The invention is not limited to the particular details of the apparatus depicted and other modifications and applications are contemplated. Certain other changes may be made in the above described apparatus without departing from the true spirit and scope of the invention herein involved. It is intended, therefore, that the subject matter in the above depiction shall be interpreted as illustrative and not in a limiting sense.

What is claimed is:

1. A circuit arrangement for calculating matrix operations of submatrices, comprising:

a matrix multiplier and a recursive accumulator, which is connected downstream from said matrix multiplier;

first and second input terminals, and one input/output terminal;

said matrix multiplier having the following units: dual-port first and second memory units for storage of input submatrices, the first memory unit being connected to the first input terminal and the second memory unit being connected to the input/output terminal, and the first memory unit having k independent memories, where k is a whole number greater than one;

a systolic arrangement, which is connected to the first and second memory units, said systolic arrangement having k multipliers connected to k series-connected adders, respectively, for multiplication and addition/substraction of input submatrices, the first memory unit storing rows of an input submatrix in k separate submemory units, the k multipliers connected via k supply lines, respectively, to the k separate sub-memory units, respectively, means for controlling said k multipliers such that each multiplier, with the associated series-connected adder successively, and in a synchronized manner, carries out a multiplication of a matrix element from a corresponding row of the first memory unit with a matrix element of a row of a submatrix from the second memory element, the k series-connected adders also connected to form an adder chain wherein results of said multiplication of matrix elements are accumulated to form product sums;

a first final adder of a carry-select type connected at an end of said adder chain; and

the recursive accumulator having the following units: third and fourth memory units, each of which being connected to the second input terminal, the third memory unit storing weightings and the fourth memory unit storing coefficients;

a further multiplier for at least one of multiplying product sums in the first final adder by the coefficients and squaring the product sums, said further multiplier connected to one of an output of the first final adder and an output of the fourth memory unit;

a further adder, which is connected downstream from the further multiplier, for component-based addition of product sums with the weightings from the third memory;

a second final adder, which is connected downstream from the further adder and is of a carry-select type; and

a recursive loop via the further adder, the second final adder or a comparator, a shifter and an overflow-monitoring circuit, and via the third memory unit for component-based MIN/MAX comparison of resulting submatrices with a submatrix which is present at the second input terminal, and for overflow regulation of the weightings which are stored in the third memory unit, said recursive loop formed by an output of the further adder being connected to one of an input of the second final adder and an input of the comparator, an input of the third memory unit being connected via the shifter and the overflow-monitoring circuit to one of an output of the second final adder and an output of the comparator, and an output of the third memory unit being connected to an input of the further adder.

2. A circuit arrangement for calculating matrix operations of submatrices, comprising:

a matrix multiplier and a recursive accumulator, which is connected downstream from said matrix multiplier;

a first input terminal and a third input terminal;

said matrix multiplier having dualport first and second memory units for storage of input submatrices, the first memory unit being connected to the first input terminal and the second memory unit being connected to the third input terminal, and the first memory unit having k independent memories, where k is a whole number greater than one;

said matrix multiplier having a systolic arrangement, which is connected to the first and second memory units, said systolic arrangement having k multipliers connected to k series-connected adders, respectively, for multiplication and addition/subtraction of input submatrices, the first memory unit storing rows of an input submatrix;

said matrix multiplier having means for controlling said k multipliers such that each of said k multipliers, with an associated series-connected adder, successively and in a synchronized manner, carries out a multiplication of a matrix element from a corresponding row of the first memory unit with a matrix element of a column of a submatrix from the

second memory element, the k series-connected adders being connected to form an adder chain wherein results of said multiplication of matrix elements are accumulated to form product sums; and

said matrix multiplier having a first final adder connected to an end of said adder chain.

3. A circuit arrangement for calculating matrix operations of submatrices, comprising:

a matrix multiplier and a recursive accumulator, which is connected downstream from said matrix multiplier;

a first input terminal, a second input terminal, and a third input terminal;

said matrix multiplier having dualport first and second memory units for storage of input submatrices, the first memory unit being connected to the first input terminal and the second memory unit being connected to the third input terminal, and the first memory unit having k independent memories, where k is a whole number greater than one;

said matrix multiplier having a systolic arrangement, which is connected to the first and second memory units, said systolic arrangement having k multipliers connected to k series-connected adders, respectively, for multiplication and addition/subtraction of input submatrices, the first memory unit storing rows of an input submatrix;

said matrix multiplier having means for controlling said k multipliers such that each of said k multipliers, with an associated series-connected adder, successively and in a synchronized manner, carries out a multiplication of a matrix element from a corresponding row of the first memory unit with a matrix element of a column of a submatrix from the second memory element, the k series-connected adders being connected to form an adder chain wherein results of said multiplication of matrix elements are accumulated to form product sums;

said matrix multiplier having a first final adder connected to an end of said adder chain;

the recursive accumulator having a third memory unit connected to the second input terminal, the third memory unit storing accumulation results;

the recursive accumulator having a further adder, which is connected downstream from the first final adder of the matrix multiplier, for component-based addition of said product sums with contents of the third memory;

the recursive accumulator having a second final adder, which is connected downstream from the further adder, results of the second final adder being stored in the third memory unit; and

the recursive accumulator having a recursive loop via the further adder, the second final adder and via the third memory unit for component-based accumulation of resulting submatrices and means to perform accumulation with a submatrix which is present at the second input terminal, said recursive loop formed by connection of the further adder to the second final adder, by connection of said second final adder to said third memory unit, and by connection of said third memory unit connected said further adder.

4. The circuit arrangement as claimed in claim 3, wherein said circuit arrangement further comprises;

a fourth memory unit for storing coefficients, said fourth memory unit being connected to the second input terminal;

a further multiplier for multiplying product sums in the first final adder by said coefficients, said further multiplier being connected to an output of the first final adder.

5. The circuit arrangement as claimed in claim 3, wherein for a component-based MIN/MAX comparison, said circuit arrangement further comprises:

a fourth memory unit connected to the second input terminal, the fourth memory unit storing accumulation results; and

a further multiplier for squaring accumulation results in the first final adder, said further multiplier being connected between an output of the first final adder and an input of the further adder, said further multiplier also connected to said fourth memory unit.

6. The circuit arrangement as claimed in claim 3, wherein said circuit arrangement further comprises:

a fourth memory unit for storing coefficients, said fourth memory unit being connected to the second input terminal;

a further multiplier for multiplying product sums in the first final adder by said coefficients, said further multiplier being connected to an output of the fourth memory unit.

* * * * *

US005325215A

United States Patent [19]

Shibata et al.

[11] Patent Number: 5,325,215

[45] Date of Patent: Jun. 28, 1994

[54] MATRIX MULTIPLIER AND PICTURE TRANSFORMING CODER USING THE SAME

[75] Inventors: Koichi Shibata, Hachiouji; Masaaki Takizawa, Suginami, both of Japan

[73] Assignee: Hitachi, Ltd., Tokyo, Japan

[21] Appl. No.: 810,173

[22] Filed: Dec. 19, 1991

[30] Foreign Application Priority Data

Dec. 26, 1990 [JP] Japan 2-406984

[51] Int. Cl.[5] H04N 1/40

[52] U.S. Cl. 358/479; 358/433; 358/471; 358/426; 382/56

[58] Field of Search 358/426, 427, 261.1, 358/261.2, 261.3, 432, 433, 448, 460, 465, 471, 479; 382/56; 341/50; 364/725, 726, 754, 736

[56] References Cited

U.S. PATENT DOCUMENTS

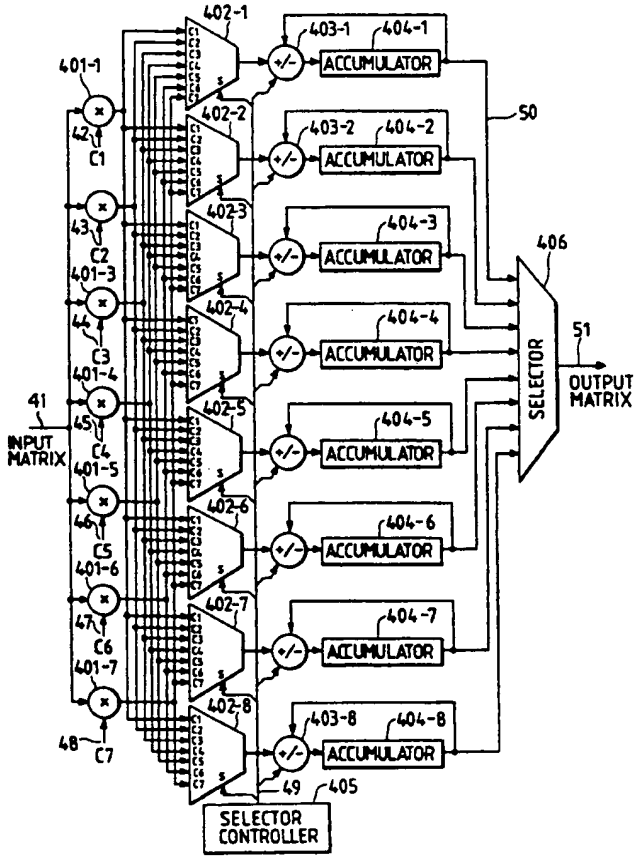
4,774,587	9/1988	Schmitt	358/426
4,922,273	5/1990	Yonekawa et al.	382/56
5,170,264	12/1992	Saito et al.	358/426
5,187,755	2/1993	Aragaki	382/56

Primary Examiner—Edward L. Coles, Sr.
Assistant Examiner—Jerome Grant, III
Attorney, Agent, or Firm—Antonelli, Terry, Stout & Kraus

[57] ABSTRACT

A matrix multiplier used in a picture transforming coder is provided for multiplying an input signal matrix by a coefficient matrix configured to comprise as many constant multipliers as absolute values of coefficients in the transform coefficient matrix for handling the signal matrix as common multiplicands, a plurality of selectors for selecting values necessary for computing elements of the matrix product from multiplication results output by the constant multipliers and a respective accumulator assigned to each of the selectors for accumulating the selected values to finally provide an element of the matrix product. As a result, since the processing can be done by merely performing as many fixed multiplications as absolute values of the coefficients in the transform coefficient matrix, the multipliers can be implemented as a relatively simple combination of adders, allowing the entire size of a picture information transforming coder or the like to be reduced.

6 Claims, 7 Drawing Sheets



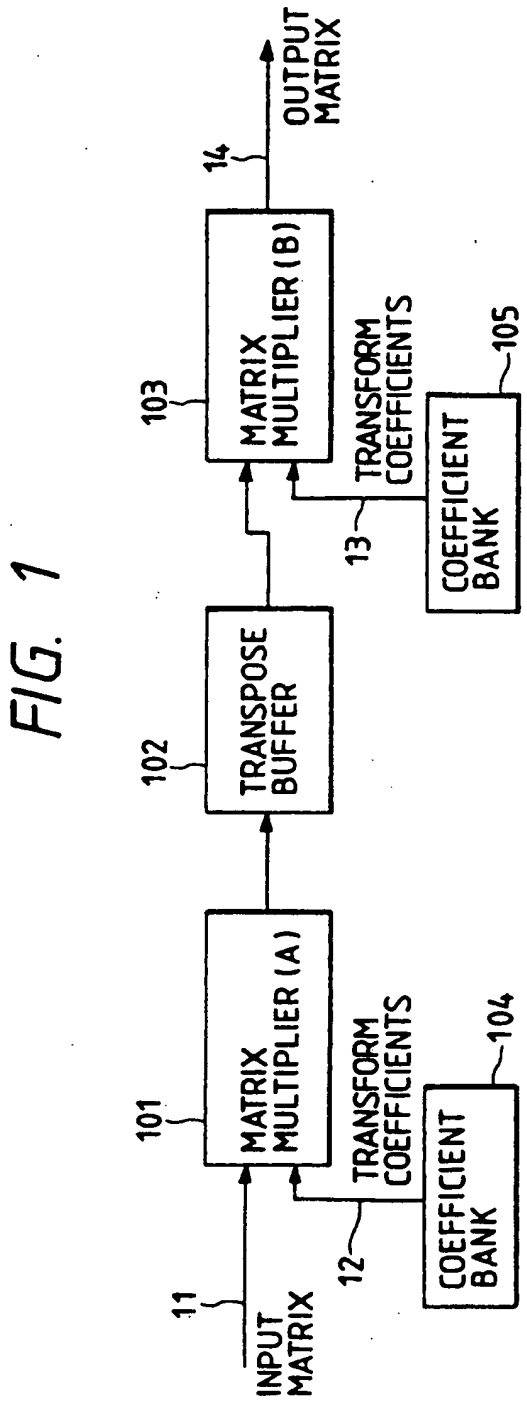

INPUT MATRIX
11
MATRIX MULTIPLIER (A)
101
TRANSFORM COEFFICIENTS
12
COEFFICIENT BANK
104
TRANSPOSE BUFFER
102
MATRIX MULTIPLIER (B)
103
TRANSFORM COEFFICIENTS
13
COEFFICIENT BANK
105
14
OUTPUT MATRIX

FIG. 1

FIG. 2 PRIOR ART

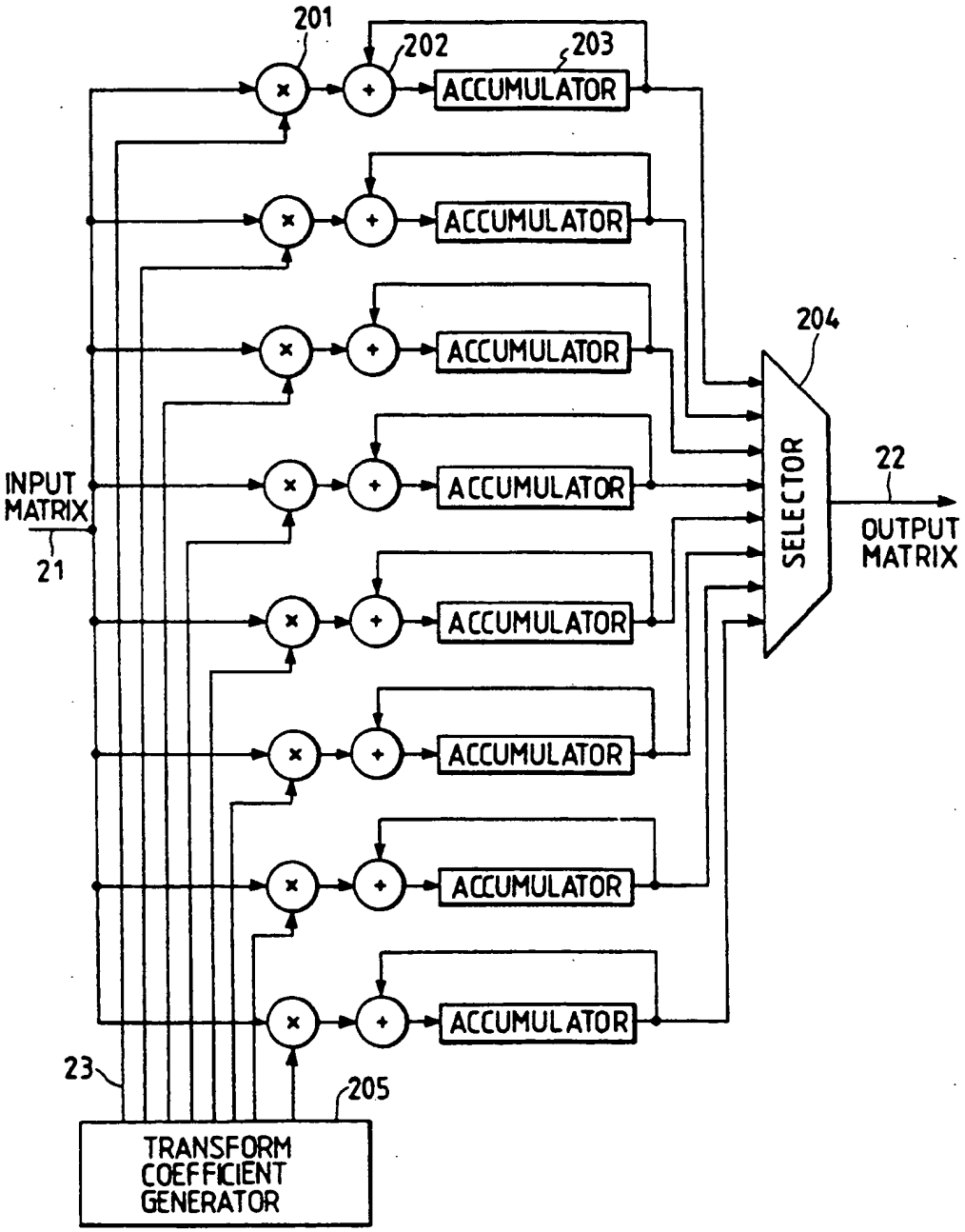


FIG. 3 PRIOR ART

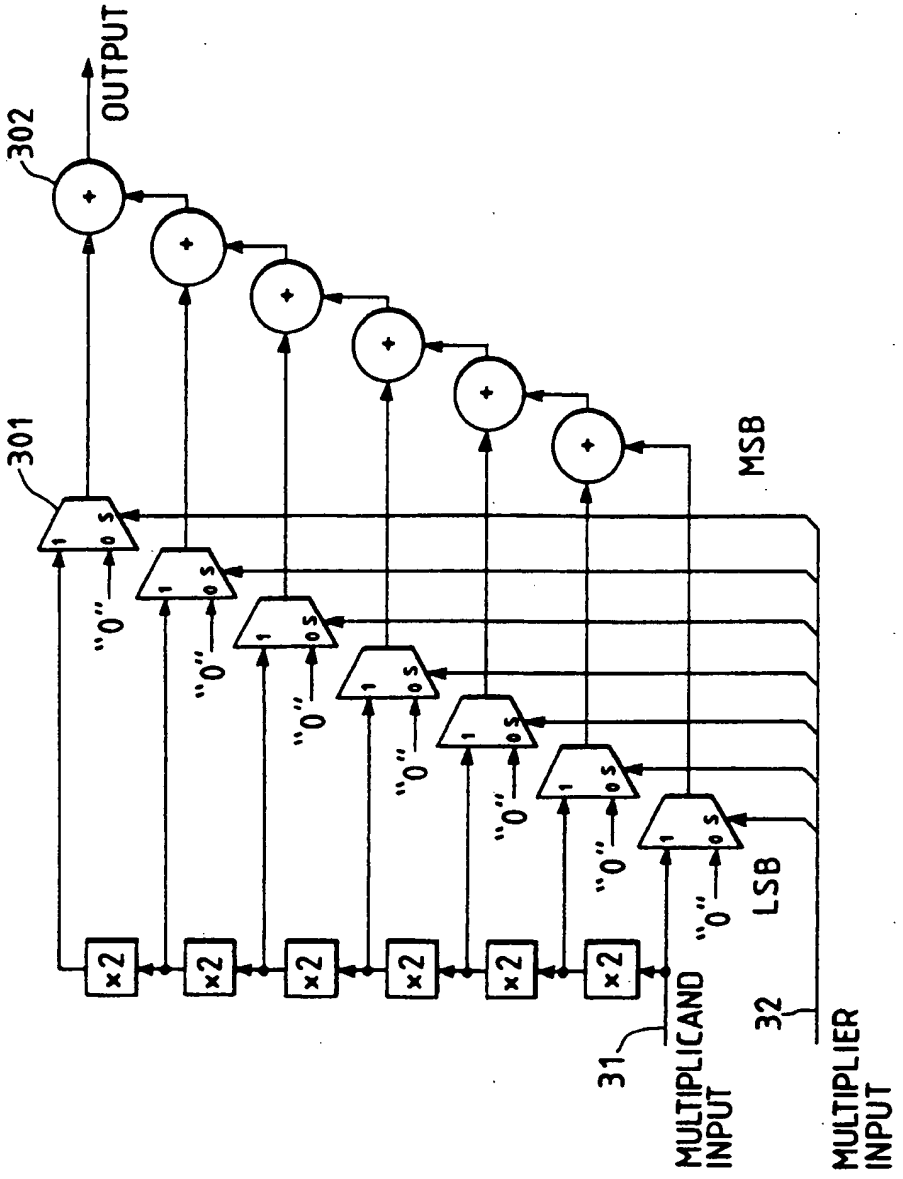


FIG. 4

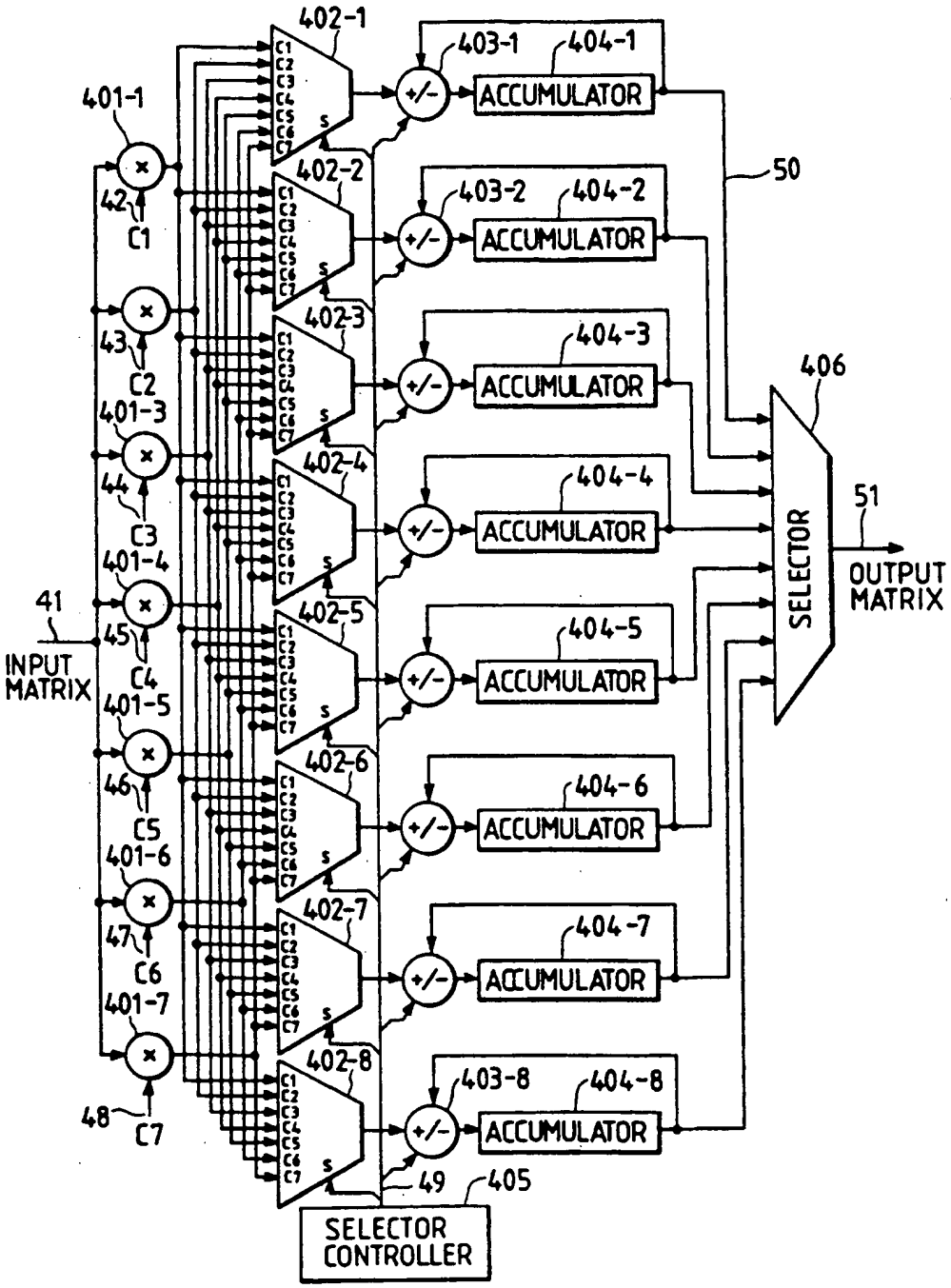

402-1
403-1
404-1
ACCUMULATOR
401-1
42
C1
402-2
403-2
404-2
ACCUMULATOR
50
402-3
404-3
ACCUMULATOR
406
43
C2
401-3
402-4
404-4
ACCUMULATOR
51
44
C3
401-4
SELECTOR
OUTPUT MATRIX
41
402-5
404-5
ACCUMULATOR
INPUT MATRIX
45
C4
401-5
402-6
404-6
ACCUMULATOR
46
C5
401-6
402-7
404-7
ACCUMULATOR
47
C6
401-7
402-8
403-8
404-8
ACCUMULATOR
48
C7
49
405
SELECTOR CONTROLLER

FIG. 5

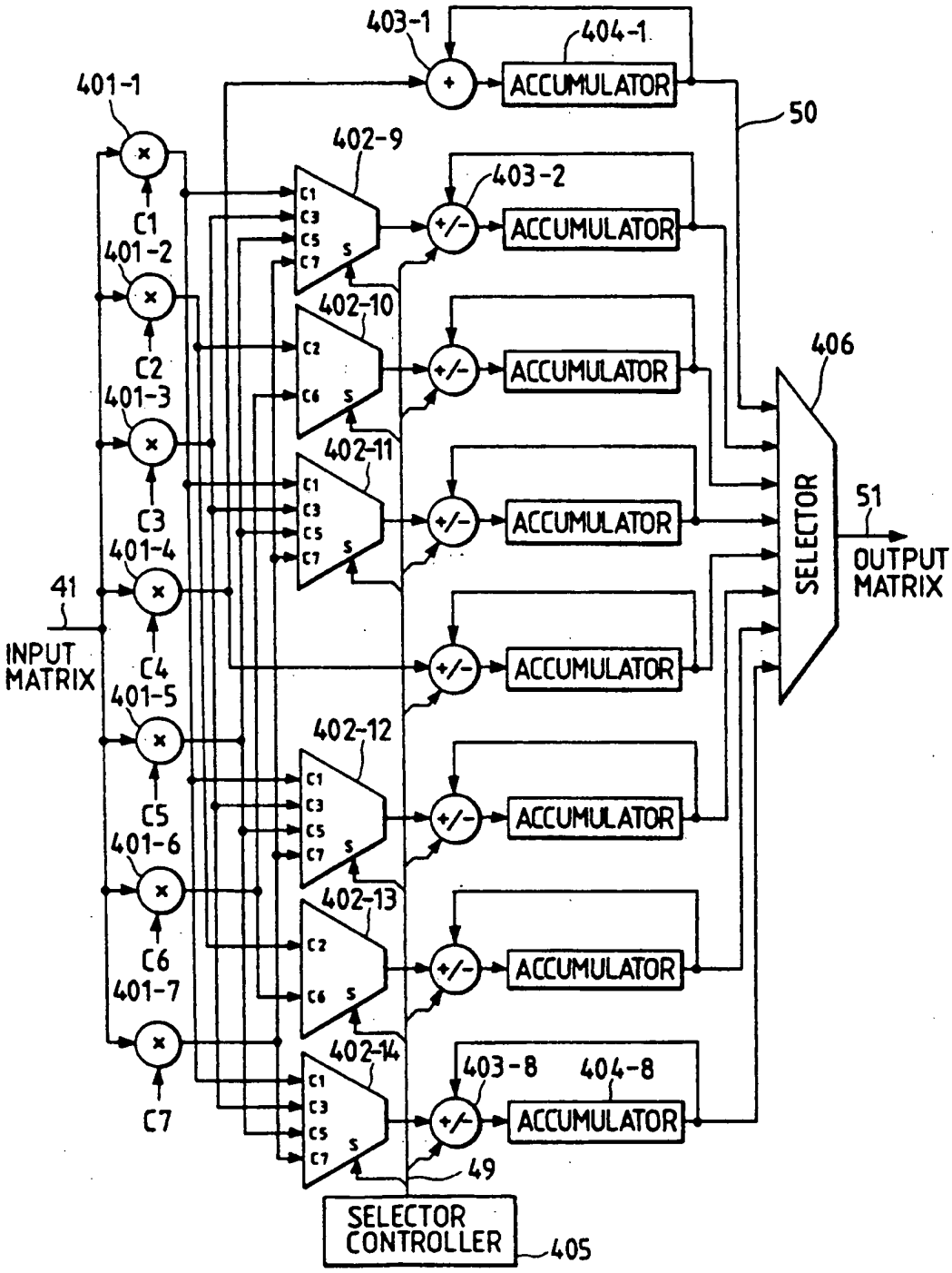

403-1
404-1
ACCUMULATOR
401-1
50
402-9
403-2
ACCUMULATOR
C1
401-2
402-10
406
C2
401-3
ACCUMULATOR
402-11
C3
401-4
51
41
SELECTOR
OUTPUT MATRIX
INPUT MATRIX
C4
401-5
ACCUMULATOR
402-12
C5
401-6
ACCUMULATOR
402-13
C6
401-7
ACCUMULATOR
402-14
403-8
404-8
C7
ACCUMULATOR
49
SELECTOR CONTROLLER
405

FIG. 6
PRIOR ART

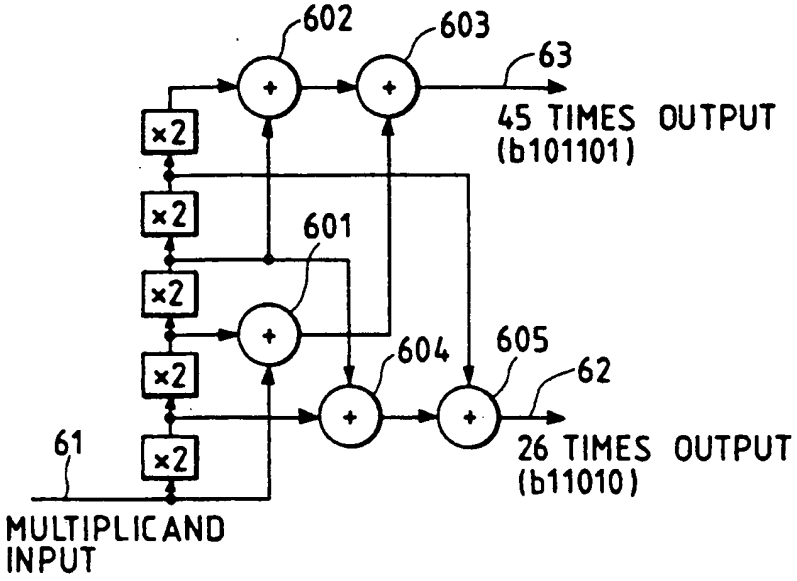
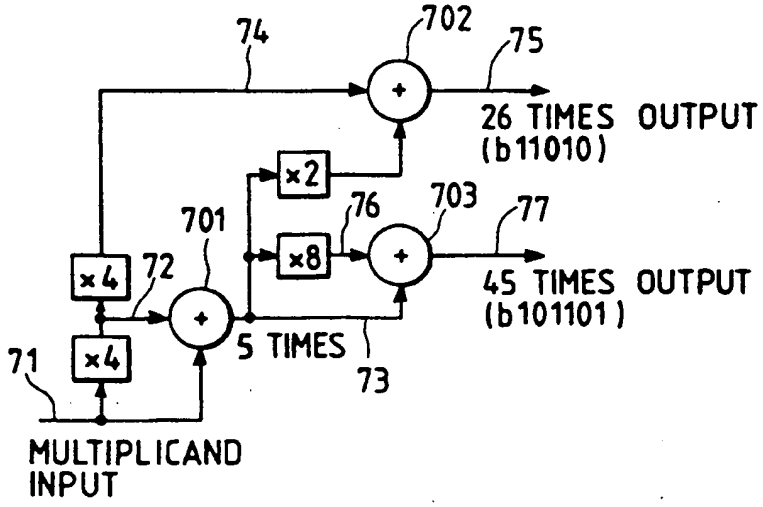


FIG. 7



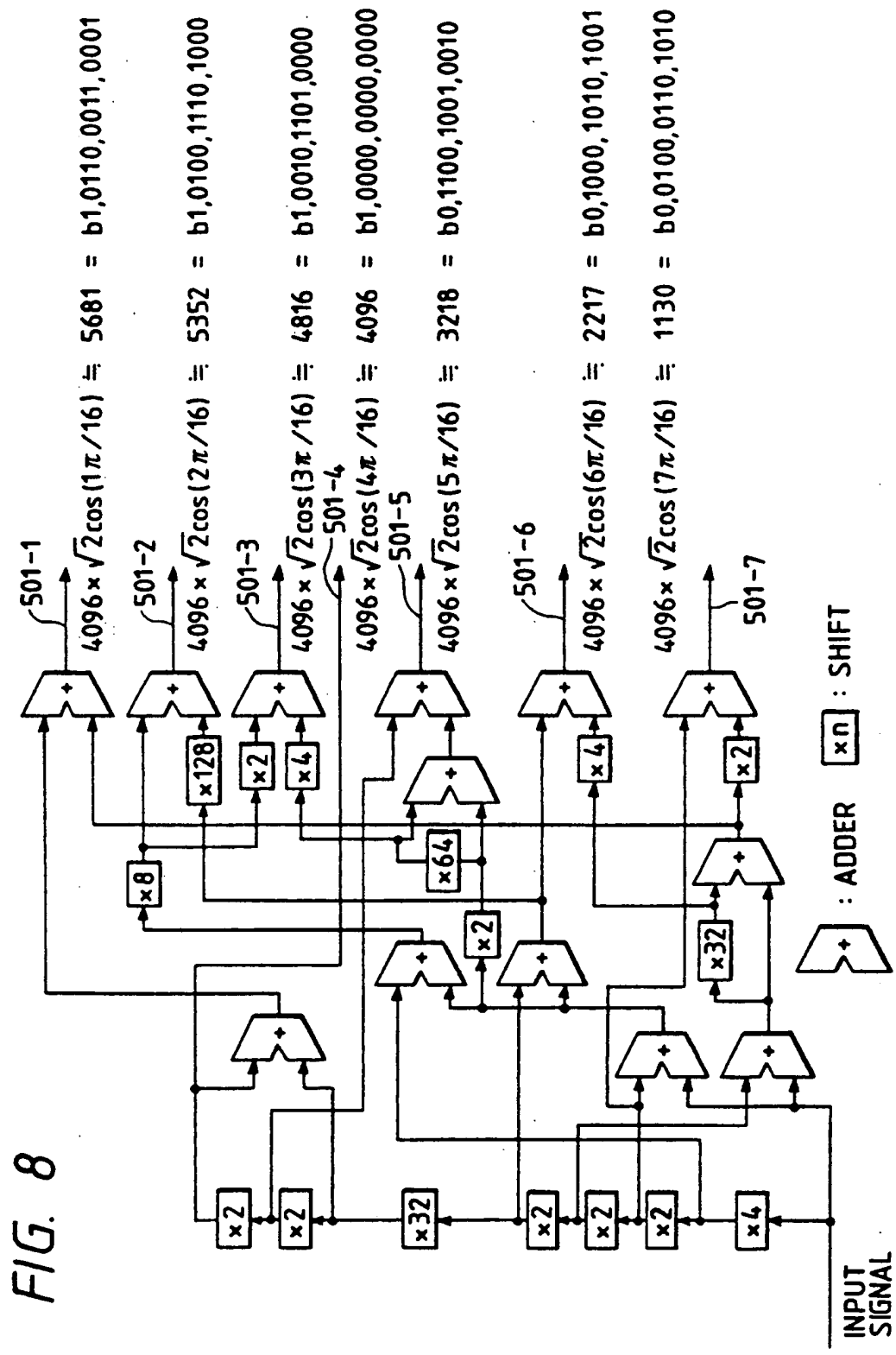

501-1
501-2
501-3
501-4
501-5
501-6
501-7
4096 × √2cos(1π/16) ≒ 5681 = b1,0110,0011,0001
4096 × √2cos(2π/16) ≒ 5352 = b1,0100,1110,1000
4096 × √2cos(3π/16) ≒ 4816 = b1,0010,1101,0000
4096 × √2cos(4π/16) ≒ 4096 = b1,0000,0000,0000
4096 × √2cos(5π/16) ≒ 3218 = b0,1100,1001,0010
4096 × √2cos(6π/16) ≒ 2217 = b0,1000,1010,1001
4096 × √2cos(7π/16) ≒ 1130 = b0,0100,0110,1010
×128
×2
×4
×8
×64
×2
×4
×2
×32
×2
×2
×32
×2
×2
×2
×4
INPUT SIGNAL
: ADDER
×n : SHIFT

FIG. 8

MATRIX MULTIPLIER AND PICTURE TRANSFORMING CODER USING THE SAME

BACKGROUND OF THE INVENTION

(1) Field of the Invention

The present invention relates to a matrix multiplier and a picture transforming coder using the matrix multiplier. In particular, the present invention is applicable to a coder and decoder for audio signals and standstill as well as moving picture information and relates to a matrix multiplier utilized in a transforming/coding unit for matrix multiplication of a coefficient matrix used in the coder by a signal matrix and a picture transforming coder utilizing the matrix multiplier.

(2) Description of the Prior Art

In many cases, a technique referred to hereafter as DCT (Discrete Cosine Transform) is used in an apparatus for efficiently coding a picture signal or the like. DCT is a kind of frequency transformation which is similar to Fourier transformation. For example, let input picture information be treated as a block of pixels comprising 8 horizontal rows and 8 vertical columns. Such a block of pixels is represented by a matrix [x] each element of which corresponds to a pixel. DCT results, or what is called DCT coefficients, are represented by a matrix [X]. Using transform coefficients [C] and its transposed matrix t[c], the relations between the matrices [x] and [X] are given by Equation (1). In many cases, DCT and inverse DCT processings are carried out in accordance with the matrix relations given by Equation 1.

Equation (1):

$$\text{DCT } [X]=[C]\times[]\times[{}^{T}C]$$

$$\text{Inverse DCT } []={}^{T}[C]\times[X]\times[C]$$

where t[ ] denotes a transposed matrix and the matrix [C] is expressed by the following equation:

$$[C]=\begin{pmatrix} C_{0\cdot0} & C_{0\cdot1} & C_{0\cdot2} & C_{0\cdot3} & C_{0\cdot4} & C_{0\cdot5} & C_{0\cdot6} & C_{0\cdot7} \\ C_{1\cdot0} & C_{1\cdot1} & C_{1\cdot2} & C_{1\cdot3} & C_{1\cdot4} & C_{1\cdot5} & C_{1\cdot6} & C_{1\cdot7} \\ C_{2\cdot0} & C_{2\cdot1} & C_{2\cdot2} & C_{2\cdot3} & C_{2\cdot4} & C_{2\cdot5} & C_{2\cdot6} & C_{2\cdot7} \\ C_{3\cdot0} & C_{3\cdot1} & C_{3\cdot2} & C_{3\cdot3} & C_{3\cdot4} & C_{3\cdot5} & C_{3\cdot6} & C_{3\cdot7} \\ C_{4\cdot0} & C_{4\cdot1} & C_{4\cdot2} & C_{4\cdot3} & C_{4\cdot4} & C_{4\cdot5} & C_{4\cdot6} & C_{4\cdot7} \\ C_{5\cdot0} & C_{5\cdot1} & C_{5\cdot2} & C_{5\cdot3} & C_{5\cdot4} & C_{5\cdot5} & C_{5\cdot6} & C_{5\cdot7} \\ C_{6\cdot0} & C_{6\cdot1} & C_{6\cdot2} & C_{6\cdot3} & C_{6\cdot4} & C_{6\cdot5} & C_{6\cdot6} & C_{6\cdot7} \\ C_{7\cdot0} & C_{7\cdot1} & C_{7\cdot2} & C_{7\cdot3} & C_{7\cdot4} & C_{7\cdot5} & C_{7\cdot6} & C_{7\cdot7} \end{pmatrix}$$

where

$$C_{ij}=\frac{1}{2}k(i)\times\cos\left(\frac{(2j+1)i\pi}{16}\right)$$

$$k(i)=1/\sqrt{2} \text{ for } i=0$$

$$k(i)=1 \text{ for } i=1,\ldots,7$$

FIG. 1 shows a general configuration of a DCT processing apparatus for executing DCT and inverse DCT manipulations. In the case of the DCT processing, the picture information matrix [x] is used as an input matrix 11. As for the inverse DCT processing, it is the DCT coefficient matrix [X] which is used as the input matrix 11. The input matrix 11 is multiplied by transform coefficients 12 stored in a coefficient bank 104 by means of a matrix multiplier (A) denoted by reference numeral 101. The results of the matrix multiplication are stored temporarily in a transpose buffer 102. The results stored in the transpose buffer 102 are further multiplied by transform coefficients 13 stored in a coefficient bank 105 by means of a matrix multiplier (B) denoted by reference numeral 103. The results of the matrix multiplication are obtained as an output matrix 14. In this case, the coefficient matrices 12 and 13 are the transform coefficients [C] and its transposed matrix t[C] respectively.

FIG. 2 shows a typical configuration of the conventional matrix multiplier for performing the DCT and inverse DCT manipulations. In other words, FIG. 2 shows the configuration of the matrix multiplier (A) or (B) denoted by reference numeral 101 or 103 in FIG. 1. Each multiplier 201 is used for multiplying an element on a row (or column) of an input matrix 21 by an element on a column (or row) of a transform coefficient matrix 23 coming from a transform coefficient generator 205. The result of the multiplication is stored in an accumulator 203 through an adder 202. The multiplication is repeated for all elements on each row (column) of the input matrix 21 and for all elements on each column (row) of the transform coefficient matrix 23 and the result of each multiplication is added to the contents of the accumulator 203. As all the elements are multiplied, the accumulators 203 are selected one after another by a selector 204 in order to output their contents sequentially as an output matrix 22.

A typical configuration of the multiplier 201 is shown in FIG. 3. For the Ith bit of a multiplier 32, a multiplicand 31 is arithmetic-left-shifted I times, where I=0 to n. Note that arithmetic-left-shifting the multiplicand 31 I times is equivalent to multiplying the multiplicand 31 by the Ith power of 2. If the Ith bit of the multiplier 32 is a '1' then the arithmetic-left-shifted result is selected by a selector 301 and added to an output by an adder 302. If the Ith bit of the multiplier 32 is a '0' however the arithmetic-left-shifted result is not selected by the selector 301 and, thus, not added to the output by tile adder 302.

SUMMARY OF THE INVENTION

In the configuration of the matrix multiplier shown in FIG. 1, as many multipliers 201 as rows (or columns) of the transform coefficients are required. In addition, the transform coefficients which are used as multipliers in the multipliers 201 may vary from row to row (or column to column) of elements of the input matrix. Accordingly, a circuit having a large scale is required, in which case operations must be carried out at a high speed. In general, it is difficult to implement a multiplier with a small physical sized hardware. For example, in order to multiply an N-digit multiplicand by an M-digit multiplier using the typical multiplier shown in FIG. 3, it is necessary to provide N ×M units of 1-bit full adders. Let us take a video telephone conforming to the international standards as an example. In order to perform real-time processing within the same time frame as the input signal with a sufficient accuracy, 8 units of matrix multipliers are required for simultaneously multiplying an input signal matrix comprising 8 rows and 8 columns of elements with each element being a 16-bit binary number by a transform coefficient matrix comprising 8 rows and 8 columns of elements with each

element being a 14-bit binary number. If the processing is to be done by the matrix multiplier shown in FIG. 2, 1,792 (16×14×8) units of adders are required. In order to carry out both the DCT and inverse DCT processings, the number of required adders is equal to this count further multiplied by 4.

It is therefore an object of the present invention to provide a matrix multiplier for multiplying an input signal matrix by a transform coefficient matrix with a multiplying unit thereof implemented using a reduced number of circuit elements.

It is another object of the present invention to provide a low-cost picture coder used in video telephones and the like for performing DCT transformations.

In order to achieve the above objects, a multiplying unit for multiplying an input signal matrix by a transform coefficient matrix is configured in accordance with the present invention to comprise:

as many coefficient multiplying means as different absolute values of coefficients in said transform coefficient matrix for multiplying said transform coefficients by input signal values with said input signal matrix used as common multiplicands for said coefficient multiplying means;

a plurality of selectors for selecting values required for computing elements of a matrix product among multiplication results output by said coefficient multiplying means; and

an accumulating means associated with each of said selectors for accumulating multiplication results to be output as an element of said matrix product.

In addition, the coefficient multiplying means are binary multipliers configured to share common processing means. The coefficient multiplying means are used to simultaneously multiply a signal value by 2 or more transform coefficients. Every common processing means described above is used for processing each bit pattern in the binary codes common to the transform coefficients.

Effective embodiments according to the present invention include a coder for processing picture information and the like using the DCT transformation. Nonetheless, applications of the present invention are not limited to such a coder.

The matrix multiplier according to the present invention employs as many multipliers as different absolute values of elements Ci, j of a transform coefficient matrix [C]. In addition, the transform coefficients which are used as multipliers do not vary during a period of inputting a row (or column) of a multiplicand matrix. Let a matrix given by Equation (2) below be the transform coefficient matrix [C] having elements Ci, j actually used for the transformation. In Equation 2, the absolute values of the elements Ci, j are represented by C1 to C7. As such, the number of element values including negative ones is 14. Accordingly, only 7 multipliers are required for multiplying the seven coefficients, C1 to C7, by an input signal value. Multiplication results required for computing elements of a matrix product are selected by the selectors and then accumulated in the accumulating means.

Equation (2)

$$[C] = \begin{pmatrix} C4 & C4 & C4 & C4 & C4 & C4 & C4 & C4 \\ C1 & C3 & C5 & C7 & -C7 & -C5 & -C3 & -C1 \\ C2 & C6 & -C6 & -C2 & -C2 & -C6 & C6 & C2 \\ C3 & -C7 & -C1 & -C5 & C5 & C1 & C7 & -C3 \\ C4 & -C4 & -C4 & C4 & C4 & -C4 & -C4 & C4 \\ C5 & -C1 & C7 & C3 & -C3 & -C7 & C1 & -C5 \\ C6 & -C2 & C2 & -C6 & -C6 & C2 & -C2 & C6 \\ C7 & -C3 & C3 & -C1 & C1 & -C3 & C5 & -C7 \end{pmatrix}$$

where

$$C1 = \frac{1}{2}\cos\left(\frac{\pi}{16}\right),$$

$$C2 = \frac{1}{2}\cos\left(\frac{2\pi}{16}\right),$$

$$C3 = \frac{1}{2}\cos\left(\frac{3\pi}{16}\right),$$

$$C4 = \frac{1}{2}\cos\left(\frac{4\pi}{16}\right),$$

$$C5 = \frac{1}{2}\cos\left(\frac{5\pi}{16}\right),$$

$$C6 = \frac{1}{2}\cos\left(\frac{6\pi}{16}\right),$$

$$C7 = \frac{1}{2}\cos\left(\frac{7\pi}{16}\right)$$

Accordingly, for few coefficient absolute values of the transform coefficient matrix [C], the number of multipliers can be reduced, thus resulting in an apparatus with a small size.

In addition, the multipliers are configured to share common processing means for manipulating common bit patterns in the binary codes of the transform coefficients.

Each of the multipliers is used to multiply an input signal by a transform coefficient which does not vary. Accordingly, each multiplier can be implemented as a relatively simple combination of adders. By providing in advance a shared processing means for each bit pattern in the binary codes common to transform coefficients, the amount of processing can be further reduced. In order to obtain results of the matrix multiplication, only necessary outputs of the multipliers are sequentially selected and then accumulated.

The foregoing and other objects, advantages of operation and novel features of the present invention will be understood from the following detailed description when read in connection with the accompanying drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a diagram showing a general configuration of a DCT processing apparatus.

FIG. 2 is a diagram showing the configuration of the conventional matrix multiplier.

FIG. 3 is a diagram showing the configuration of the conventional multiplier.

FIG. 4 is a diagram showing the configuration of an embodiment implementing a matrix multiplier in accordance with the present invention.

FIG. 5 is a diagram showing tile configuration of another embodiment implementing a matrix multiplier in accordance with the present invention.

FIG. 6 is a diagram showing the configuration of the conventional coefficient multipliers.

FIG. 7 is a diagram showing the operational configuration of an embodiment implementing coefficient multipliers in accordance with the present invention.

FIG. 8 is a diagram showing an actual circuit configuration of the multipliers 401-1 to 401-7 employed in the matrix multiplier for manipulating the DCT processing shown in FIG. 4 or 5.

DESCRIPTION OF THE PREFERRED EMBODIMENTS

Referring to diagrams, embodiments according to the present invention are described as follows.

FIG. 4 is a diagram showing the configuration of an embodiment implementing a matrix multiplier in accordance with the present invention. The embodiment is configured to implement a circuit for executing the DCT code transformation on picture information comprising 8×8 pixels. The pixels each correspond to an element of a matrix. The elements of the matrix to be multiplied simultaneously are supplied as matrix multiplicand inputs 41 which are fed to a plurality of multipliers 401-1 to 401-7. The elements are each multiplied by transform coefficients 42 to 48 having seven different values represented by C1 to C7 in Equation (2). Only values required for computing the matrix product are selected from results obtained from the multipliers 401-1 to 401-7 by selectors 402-1 to 402-8 in accordance with the matrix product equation for the matrix of Equation (2). Note that each of the selectors 402-1 to 402-8 is associated with an element of the matrix product. The order in which the values are selected by one of the selectors 402-1 to 402-8 is determined by a select control line 49 which is programmed by a selector controller 405. It should be noted that for the sake of diagrammatical simplicity, the select control line 49 is denoted by a single line in the figure. In the actual configuration, however, each of the selectors 402-1 to 402-8 is provided with one independent select control line.

A selected value is then added to or subtracted from contents 50 of an accumulator 404 by an adder/subtractor 403. The selection as to whether an addition or subtraction is to be carried out depends upon whether the value of the transform coefficient multiplied is positive or negative.

As the matrix multiplication for an input matrix is completed, the accumulators 404 each contain an element of the matrix product. The elements of the matrix product are sequentially fetched by a selector 406 and transmitted as a matrix processing output 51. The selector 406 has the same configuration as the conventional selector shown in FIG. 2 which the reader is already familiar with. In actuality, each of the accumulators 404-1 to 404-8 is provided with a register for storing an element of the matrix product. At the completion of the matrix multiplication, the contents of each register are reset to zero. The contents of each register are output as an output matrix 81 at fixed timing. In the mean time, the multipliers 401-i, the selectors 402-j and the accumulators 404-j, where i=1 to 7 and j=1 to 8, are used to carry out the matrix multiplication for the next input multiplicand matrix in the same way as the preceding matrix multiplication.

FIG. 5 is a diagram showing the configuration of another embodiment implementing a matrix multiplier in accordance with the present invention. In the latter matrix multiplication of the DCT transformation and the former matrix multiplication of the inverse DCT transformation, the number of coefficients used in the matrix multiplications is even smaller. The fact that the coefficient count is lower allows the configuration of the selectors 402 in the embodiment to be further simplified. To be more specific, in addition to a reduced number of selectors 402, each of the selectors 402 can also be made much simpler. The matrix expressed by Equation (2) clearly indicates that the selectors 402-1 and 402-5 always select products of multiplying the transform coefficient C4. Accordingly, the selectors 402-1 and 402-5 and their select control lines 49 are not required. As for the selectors other than the 402-1 and the 402-5, their number of inputs to be selected is reduced to 4 or 2, resulting in a simplified circuit of the selector controller 405 and simplified select control lines 49.

FIG. 7 is a diagram showing the operational configuration of the coefficient multipliers 401 used in the matrix multiplier in accordance with the present invention. A multiplier for multiplying coefficients is implemented as a combination of adders such as the conventional one shown in FIG. 6. As shown in FIG. 6, a procedure for computing a 26 times output 62 and a 45 times output 63 of a multiplicand input 61 is given as follows. The binary format of the number 26 is known to be '11010' which is the sum of the 16 times, the 8 times and the 2 times. Accordingly, the 26 times output can be obtained by adding the 16 times, the 8 times and the 2 times with adders 604 and 605. Similarly, the binary format of the number 45 is known to be '101101' which is the sum of the 32 times, the 8 times, the 4 times and the 1 time. Accordingly, the 45 times output can be obtained by adding the 32 times, the 8 times, the 4 times and the 1 time with adders 601, 602 and 603. In this case, since multiplying a number by the nth power of 2 is equivalent to arithmetic-left-shifting the number, no special processing is required. Accordingly, five units of adders are used as is shown in the figure.

On the other hand, with a technique provided by the present invention, a common bit pattern in the binary codes of the coefficients is identified and assigned to a common processing means to be shared by the multipliers. For example, the binary format of the number 26 is '11010' whereas that of the number 45 is '101101' as described above. A bit pattern '101' is common to both the binary codes '11010' and '101101'. Accordingly, when computing 26 times and 45 times outputs at the same time, the 5 times calculation corresponding to the common bit pattern '101' can be carried out by a common processing means provided in advance and to be shared by the multipliers. The 26 times output is then computed by adding the product of the 5 times and the 2 times to the 16 times while the 45 times output is calculated by adding the product of the 5 times and the 8 times to the 5 times. In this way, the processing can be simplified. To be more specific, a 4 times value 72 of a multiplicand input 71 is added to the multiplicand input 71 by an adder 701 to result in a 5 times value 73 which is multiplied by 2 and then added to a 16 times value 74 of the multiplicand input 71 by using an adder 702 to

give the 26 times output 75. At the same time, the 5 times value 73 is multiplied by 8 to result in a 40 times value 76 which is added to the 5 times value 73 to give the 45 times output 77. In the embodiment shown in FIG. 7, the number of adders is thereby reduced to three.

It is obvious that the technique adopted in the above embodiment is not limited to the computation of the 26 and 45 times but can also be applied to all cases as well. In addition, any common bit pattern can always be assigned to a shared processing means as in the case with the common bit pattern '101'.

FIG. 8 is a diagram showing an actual circuit configuration of the multipliers 401-1 to 401-7 employed in the matrix multiplier for manipulating the DCT processing shown in FIG. 4 or 5. In this case, it is assumed that in the configuration shown in FIG. 8, coefficients to be multiplied by input data are alpha×C1 to alpha×C7, where alpha=4,096×2$\sqrt{2}$. Accordingly, if the value of alpha changes, the number of times the arithmetic-left-shift operation to be performed also varies. Outputs of adders 501-1 to 501-7 shown in FIG. 8 correspond to the outputs of the multipliers 401-1 to 401-7 shown in FIG. 4 or 5. With the configuration of the multipliers shown in FIG. 8, the number of adders can be reduced to only 13. Therefore, in order to give a 16-bit processing accuracy, only 178 (=13×16) units of adders are needed. When compared to the 1,792 units of adders required in the multipliers adopting the conventional technique, the number 178 clearly means that the DCT transformation can be carried out to achieve the same processing accuracy using only one tenth as much of the conventional hardware.

The above embodiments are described mainly for applications for manipulating 8×8 elements. It should be noted, however, that applications of the present invention are not limited to the processing of 8×8 elements. The present invention is particularly effective when applied to a code transforming apparatus for handling picture information even though it is not restricted to such an application.

In the matrix multiplier provided by the present invention, the multipliers for outputting elements of a matrix product have a configuration in which coefficients are treated as multiplier inputs. In addition, the number of multipliers is independent of the number of elements in a row or column of the input multiplicand matrix and determined only by the number of absolute values of the coefficients in the transform coefficient matrix. Accordingly, the number of multipliers can be reduced. As a result, the invention allows a matrix multiplier having a small size and a low cost to be implemented. In particular, the effects of the present invention on the implementation and popularization of a transforming coder for picture information are substantial since the reduction in apparatus size and cost is a very important requirement.

We claim:

1. A picture transforming coder for efficiently coding a picture signal comprising:

a first matrix multiplier for performing matrix processing on an input signal and a first transform coefficient matrix; and

a second matrix multiplier for performing matrix processing on outputs of said first matrix multiplier and a second transform coefficient matrix which is the transposed matrix of said first transform coefficient matrix, said second matrix multiplier outputting matrix processing results as a coded picture signal;

wherein said first matrix multiplier includes:

a plurality of constant multiplying means for receiving said input signal in predetermined block units, handling each of said block units as a multiplicand matrix and multiplying each element of said multiplicand matrix by each element of said first transform coefficient matrix,

a plurality of selectors for selecting necessary output values from multiplication results output by said constant multiplying means,

a control means for controlling said selectors by following predetermined rules established in accordance with said first transform coefficient matrix, and

a plurality of accumulating means for storing elements of the matrix product by accumulating said necessary output values with each of said accumulating means assigned to one of said selectors.

2. A picture transforming coder according to claim 1, wherein said input signal received in said block units is two-dimensional information on a plurality of pixels adjacent to each other;

said first transform coefficient matrix is a DCT coefficient matrix; and

said second transform coefficient matrix is the transposed matrix of said DCT coefficient matrix.

3. A picture transforming coder according to claim 1, wherein said constant multiplying means is configured to include a common numerical processing means shared by a first numerical processing means for multiplying a first coefficient of said first transform coefficient matrix by a matrix element of said multiplicand matrix, a second numerical processing means for multiplying a second coefficient of said first transform coefficient matrix by said matrix element and any other numerical processing means for multiplying other coefficients of said first transform coefficient matrix by said matrix element so that said constant multiplying means are capable of multiplying two or more coefficients of said first transform coefficient matrix by an element of said multiplicand matrix; and

said common numerical processing means is used for handling a bit pattern common to the binary codes of said first coefficient, said second coefficient and said other coefficients being multiplied.

4. A matrix multiplier applicable to an apparatus for performing predetermined coded on an input signal comprising:

a plurality of constant multiplying means, connected to receive said input signal in predetermined block units, for each handling a respective one of said block units as a multiplicand matrix and for multiplying each element of said multiplicand matrix by a respective of a coefficient matrix;

a plurality of selectors each for selecting necessary output values from multiplication results output by said plurality of constant multiplying means;

control means for controlling said selectors by following predetermined rules established in accordance with said coefficient matrix; and

a plurality of accumulating means each storing elements of a matrix product by accumulating respective ones of said necessary output values received from a respective one of said selectors.

5. A matrix multiplier according to claim 4, wherein said input signal received in said block units is two-di-

mensional information on a plurality of pixels adjacent to each other; and

said coefficient matrix is a DCT coefficient matrix of the transposed matrix of said DCT coefficient matrix.

6. A matrix multiplier applicable to an apparatus for performing predetermined coding on an input signal comprising:

a plurality of constant multiplying means for receiving said input signal in predetermined block units, handling each of said block units as a multiplicand matrix and multiplying each element of said multiplicand matrix by each of a coefficient matrix;

a plurality of selectors for selecting necessary output values from multiplication results output by said constant multiplying means, a control means for controlling said selectors by following predetermined rules established in accordance with said coefficient matrix; and

a plurality of accumulating means for storing elements of the matrix product by accumulating said necessary output values with each of said accumulating means assigned to one of said selectors;

wherein said constant multiplying means is configured to include a common numerical processing means shared by a first numerical processing means for multiplying a first coefficient of said coefficient matrix by a matrix element of said multiplicand matrix, a second numerical processing means for multiplying a second coefficient of said coefficient matrix by said matrix element and any other numerical processing means for multiplying other coefficients of said coefficient matrix by said matrix element so that said constant multiplying means are capable of multiplying two or more coefficients of said coefficient matrix by an element of said multiplicand matrix; and wherein

said common numerical processing means is used for handling a bit pattern common to the binary codes of said first coefficient, said second coefficient and said other coefficients being multiplied.

* * * * *